M T S

The Michigan Terminal System

Volume 14: 360/370 Assemblers in MTS

May 1983

Updated September 1986 (Update 1)

The University of Michigan Computing Center
Ann Arbor, Michigan

****
*
* This obsoletes the August 1978 edition. *
*
****

DISCLAIMER

The MTS Manual is 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 Computing Center Newsletter, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed processor computer 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 Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center 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 MTS volumes are fully up to date.

Volume 1: The Michigan Terminal System, January 1984
Volume 2: Public File Descriptions, April 1982
Volume 3: System Subroutine Descriptions, April 1981
Volume 4: Terminals and Networks in MTS, March 1984
Volume 5: System Services, May 1983
Volume 6: FORTRAN in MTS, October 1983
Volume 7: PL/I in MTS, September 1982
Volume 8: LISP and SLIP in MTS, June 1976
Volume 9: SNOBOL4 in MTS, September 1975
Volume 10: BASIC in MTS, December 1980
Volume 11: Plot Description System, August 1978
Volume 12: PIL/2 in MTS, December 1974
Volume 13: The Symbolic Debugging System, September 1985
Volume 14: 360/370 Assemblers in MTS, May 1983
Volume 15: FORMAT and TEXT360, April 1977
Volume 16: ALGOL W in MTS, September 1980
Volume 17: Integrated Graphics System, December 1980
Volume 18: The MTS File Editor, August 1985
Volume 19: Tapes and Floppy Disks, February 1983
Volume 20: Pascal in MTS, December 1985
Volume 21: MTS Command Extensions and Macros, April 1986

Other volumes are in preparation. 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. Volume 1, for example, introduces the user to

Page Revised September 1986 May 1983

MTS and describes in general the MTS operating system, while 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

May 1983

PREFACE TO VOLUME 14

The May 1983 revision reflects the changes that have been made to MTS since August 1978. Some of these changes were described Update 1 (October 1979), Update 2 (April 1980), and Update 3 (January 1981) and are incorporated into this edition.

Since January 1981, further changes have been made to the Message Macros and the Structured Programming Macros and are described herein. In addition, the descriptions of the *ASMG and *ASMT assemblers have been removed since they are no longer actively supported by the Computing Center. These descriptions are available through the *GENDOC program.

The sections "IOH," "Extensions to the Amdahl 470 Operations," and "Extensions to the System/360 Model 67 Operations" have been moved into Volume 14 from MTS Volume 5, System Services.

May 1983

Contents

May 1983

360/370 ASSEMBLERS IN MTS

This edition contains information for using those translators which translate languages that are essentially 360 or 370 machine languages. The translators described are

Assembler H

ASSIST

The August 1978 edition describes

Assembler G

TSS Assembler

Descriptions of these assemblers are also available through the *GENDOC program. The Assembler G and TSS Assembler translators are no longer actively supported by the Computing Center.

All of these translators use a traditional assembler format (one instruction per line, with each line divided into a label field, an opcode field, an operands field, and a comments field) and translate approximately the same source language.

LANGUAGE REFERENCES

The language reference manual for Assembler G, Assembler H, and ASSIST is the IBM publication, IBM System/360 Operating System Assembler Language, form number GC28-6514. The language reference manual for the TSS Assembler is the IBM publication, IBM System/360 Time-Sharing System Assembler Language, form number GC28-2000.

For a description of the individual machine instructions, see the IBM publication, IBM System/370 Principles of Operation, form number GA22-7000. In addition, certain special instructions that are available on the IBM 360/67 and on the Amdahl 470 and 5860 computers when running in MTS are described in this volume.

SUPPLEMENTARY MATERIAL

This volume also contains the descriptions of the macros provided by the Computing Center for assembly language programmers. For those using Assembler G and Assembler H, the macros will be found in the public file

*SYSMAC; for those using the TSS Assembler, the macros will be found in the public file *ASMTSYSMAC. The majority of these macros aid the programmer in providing calling sequences for subroutines of the same name. The macro descriptions found in this volume assume that the user is familiar with the related subroutine descriptions which appear in MTS Volume 3, System Subroutine Descriptions.

In addition, this volume contains public file descriptions for certain programs of interest to assembly language users. These public file descriptions also appear in MTS Volume 2, Public File Descriptions.

For information on the detailed format of object records which are produced by the translators described in this volume, see the section "The Dynamic Loader" in MTS Volume 5, System Services.

ASSEMBLER H

The following section describes the use of the IBM program product Assembler H which has been adapted for use in MTS.

Several of the extensions to Assembler H described in the subsection "Assembler H Extensions" are adapted from changes made to Assembler H by Gregory J. Mushial at the Stanford Linear Accelerator Center. In particular items 20, 21, and 25 are taken from Mr. Mushial's changes as well as the &SYSNEST system set symbol, the REL2 parameter, the nullified USING message (IEV056), and the USING map that appears as part of the listing page header. The descriptions of the above changes are adapted from SLAC Computing Services User Note 100 by Mr. Mushial.

Page Revised September 1986 May 1983

*ASMH

Contents: The initial object module of the MTS Assembler H.

Purpose: To assemble System/370 assembly language programs.

Use: The assembler is invoked by the $RUN command.

Logical I/O Units Referenced:

SCARDS - the source program to be assembled.
SERCOM - assembler diagnostics.
SPRINT - listing and reference tables.
SPUNCH - the resulting object module.
0 - a library of macro definitions.
If 0 is not assigned, then *SYSMAC will be used.
1 - the resulting object module.
2-10 - additional libraries of macro definitions.

If the assembler is being called from within a program, the logical I/O units may differ. See the subsection "Invoking ASMH from a Program."

| Description: This assembler is the IBM program product Assembler H, Version 2 (program number 5668-962) modified for use in MTS. The language accepted by ASMH is given in the IBM publication, Assembler H Version 2 Application Programming: Language Reference (form GC26-4037). Extensions to the language are discussed in the subsection "Assembler H Extensions." The error messages produced by ASMH are listed and explained in the IBM publication, Assembler H Version 2 Application Programming: Guide (form SC26-4036). Additions and changes to these messages are described in the subsection "Assembler H Messages."

ASSEMBLER OPTIONS

The programmer may specify the following options in the PAR field of the $RUN command. The entries may appear in any order and, if any are missing, a standard default will be assumed. Commas are accepted as parameter delimiters and keyword operands may be delimited on the left with equal signs or may be enclosed in parentheses (except XREF which may not be used with an equal sign).

No abbreviations are allowed for the options. The default forms are given at the right of each option name.

ALGN or ALIGN / NOALGN or NOALIGN Default: ALGN

The ALGN option specifies that all alignment errors are to be flagged. The NOALGN option specifies that only alignment errors involving the fetching of instructions (e.g., BC or LPSW) are to be flagged.

BATCH or MULT / NOBATCH or NOMULT Default: NOBATCH

With the BATCH option, the assembler processes a stream of assemblies, the last assembly being terminated by an end-of-file. See the following subsection, "BATCH Option." With the NOBATCH option, the assembler processes only one assembly and then returns to the calling program.

| CALIGN=n / NOCALIGN Default: NOCALIGN
|
| The CALIGN option controls the placement of fields in generated statements. NOCALIGN, the default, prints the operation, operand, and comment fields in the same columns they were found in the source statement, if possible. CALIGN=0 prints the operation and operand fields in columns 10 and 16, respectively. In addition, CALIGN=n (16<n≤72) positions comments in generated statements at the specified column.

DECK / NODECK Default: DECK

The DECK option specifies that the object module is produced and written on the logical I/O unit SPUNCH. With the NODECK option, no object module is written on SPUNCH.

ESD / NOESD Default: ESD

The ESD option specifies that the external symbol dictionary is listed on the logical I/O unit SPRINT. The NOESD option suppresses the listing of the external symbol dictionary. See also the LIST option which controls SPRINT output.

EXTEN / NOEXTEN Default: EXTEN

The EXTEN option specifies that certain extensions to the OS Assembler H are allowed. See the subsection "Assembler H Extensions." With the NOEXTEN option, strict compatibility with the OS Assembler H is observed.

FLAG(n) or MSGLEVEL(n) Default: FLAG(0)

The FLAG option specifies the level below which error diagnostics will not be printed. See the IBM publication, OS Assembler H

Messages (form SC26-3770), for the levels of diagnostic messages. The default is FLAG(0) which prints all diagnostic messages.

LINECNT(n) or LINECOUNT(n) Default: LINECNT(55)

The LINECNT option specifies the number of lines to be printed between the headings in the source listing. The limits are from 1 to 32767 lines.

LIST / NOLIST Default: See text

With the LIST option, SPRINT output from the assembler will occur as specified by the ESD, RLD, and XREF options. This is the default in batch mode and in conversational mode, if SPRINT is explicitly assigned or has defaulted to something other than the terminal. With the NOLIST option, all SPRINT output is suppressed including the header, source listing, and summary. In this case, the ESD, RLD, and XREF options are ignored. This is the default in conversational mode if SPRINT has not been explicitly assigned and is defaulted to the terminal. The recommended method of obtaining SPRINT output at the terminal is to specify SPRINT=*MSINK*.

LOAD or OBJECT / NOLOAD or NOOBJECT Default: NOLOAD

The LOAD option specifies that the object module is produced and written on logical I/O unit 1. With the NOLOAD option, no object module is written on logical I/O unit 1.

| MACXREF / NOMACXREF Default: NOMACXREF
|
| The MACXREF option prints a cross-reference listing of the macros and copy sections used from various libraries. The NOMACXREF option suppresses this listing.
|
| NUM(LEFT) or NUM / NUM(RIGHT) / NONUM Default: NUM(LEFT)
|
| The NUM option specifies whether line numbers are printed on the assembly listing and on SERCOM error messages. NUM(LEFT) specifies that line numbers are printed between the statement number and source statement fields on the listing and included in error messages written to SERCOM. NUM is equivalent to NUM(LEFT). NUM(RIGHT) specifies that line numbers are printed on the right side of the listing and on SERCOM. NONUM specifies that line numbers are not printed on either the listing or SERCOM. In this case, the space in the listing between the statement number and the source statement is closed up. NUM={LEFT|RIGHT} notation may be used instead of the parentheses when specifying NUM in the PAR field.
|
| With NONUM in effect, the listing is truly OS-compatible, with a maximum line length of 121 bytes. With either NUM(LEFT) or NUM(RIGHT) in effect, the listing has a maximum line length of 133. This information may be useful when writing listings to tape.

PEXIT=*PEXIT

If the PEXIT parameter is specified, the *PEXIT listing postprocessor will be invoked to improve the readability of the source listing. This feature is most useful when using the structured-programming macro package. See the description of *PEXIT in this volume for further details on the postprocessor.

REL2 / NOREL2 Default: NOREL2

The REL2 option suppresses the error message generated by the assembler when a halfword relocatable adcon is used.

RENT / NORENT Default: NORENT

The RENT option specifies that the assembler will check for statements with reentrancy violations (caused by instructions that change a location within a CSECT). With the NORENT option, this check is not made.

RLD / NORLD Default: NORLD

The RLD option specifies that the relocation dictionary is listed on the logical I/O unit SPRINT. The NORLD option suppresses the listing of the relocation dictionary. See also the LIST option which controls SPRINT output.

Page Revised September 1986 May 1983

SYSPARM(n) Default: See text

The SYSPARM option specifies the character string value of the system variable symbol &SYSPARM. If the SYSPARM option is not specified, &SYSPARM will default to a null string. Commas are not allowed unless the value of the SYSPARM parameter is enclosed in balanced parentheses or primes. If the parameter is enclosed in primes, an embedded prime must be represented by two primes. The enclosing primes or parentheses are considered as part of the value unless no equal sign is given, in which case the outer parentheses are not included. For example,

```
SYSPARM(&AB,('&XY))
```

assigns &AB,('&XY) to &SYSPARM.

TERM / NOTERM Default: See text

If the TERM option is specified, flagged lines and assembler diagnostics are listed on SERCOM. TERM is the default in conversational mode unless SPRINT has been explicitly assigned to the terminal (SPRINT=*MSINK*). If the NOTERM option is specified, the TERM option is suppressed. NOTERM is the default in batch mode.

TEST / NOTEST Default: NOTEST

The TEST option specifies that the object module includes SYM records used by the Symbolic Debugging System for program debugging. With the NOTEST option, no SYM records are produced. See MTS Volume 13, The Symbolic Debugging System, for further details on using SDS.

UMAP / NOUMAP Default: UMAP

If the UMAP option is specified, the currently active USING statements will be printed at the top of each page of the source listing. This will be truncated without warning to fit on two lines, if necessary. The statements printed are those current as of the end of the assembly of the first instruction on the new page; the first statement may not actually appear in the listing if it is, for example, an EJECT, SPACE, or TITLE statement. If NOUMAP is specified, these statements will not be printed.

XREF(FULL) or XREF / XREF(SHORT) / NOXREF Default: XREF(FULL)

The XREF(FULL) option specifies that a cross-reference table containing all symbols defined in the program (whether referenced or not) and all literals used in the program will be printed on the logical I/O unit SPRINT. The XREF(SHORT) option specifies that the above cross-reference table will be printed on SPRINT except that it will not contain unreferenced symbols. The NOXREF option suppresses the printing of the cross-reference table. See also the LIST option which controls the SPRINT output.

BATCH OPTION

If the option BATCH is specified, the assembler will accept multiple source decks from SCARDS. The decks are only delimited by the END card of the preceding deck, and the end of the batch is indicated by an end-of-file on SCARDS. The listing of each source deck is preceded by the usual header page, and the object decks appear one after another on SPUNCH and/or logical I/O unit 1.

The return code returned to the invoking program is the highest code encountered in any of the assemblies in the batch.

Note that MULT is a synonym for BATCH and NOMULT is a synonym for NOBATCH.

ASSEMBLER H EXTENSIONS

Several extensions to the language accepted by Assembler H have been made. Most of these are not available in non-MTS versions of Assembler H and, in the MTS version, are available only if the EXTEN option is specified (the default). By specifying NOEXTEN, the assembler will be fully compatible with the non-MTS version of Assembler H. The extensions are as follows:

(1) Literals can be used in EQU statements such as "A EQU =A(ABC)" and in expressions such as "IC 0,=X'010203'-1".

(2) The "*nnn" on LCLC and GBLC declarations for *ASMG is ignored. This makes the macro language almost upward compatible with Assembler G. Note that SETC symbols may be up to 255 characters long without any special declaration.

(3) Five new system set symbols are allowed in macro definitions:

&SYSCCID is the signon ID running *ASMH (this is also allowed outside a macro definition).

&SYSSTYP is the type of section from which the macro was called (CSECT, COM, or DSECT).

&SYSSTMT is the statement number from which the macro was called.

&SYSLINE is the MTS line number of the top level macro call. When used as a character string, its value is the same as the line number printed by the assembler (with no blanks). When used as an arithmetic or Boolean value, its value is the internal MTS line number as a signed integer (the internal line number is the external line number times 1000). If it is

assigned to a SETC symbol and the result is used in an arithmetic or Boolean expression, an error will result.

&SYSNEST contains the macro nesting depth where the top-level macro called from open code has a SYSNEST value of 1.

(4) A predefined, absolute symbol can be used in a SETA or SETB expression and as the value of a SETC symbol or macro parameter used in a SETA or SETB expression. For example, "&A SETA JTBLPOOL" or "AIF (LENGTH GT 256).ERROR". The symbols used must be predefined and will not be cross-referenced.

(5) The value of &SYSNDX is 5 instead of 4 digits long. This will not cause a problem with symbols that are too long since Assembler H allows symbols up to 63 characters long.

(6) Macro call operands are scanned for sublists, keywords, etc., after all substitution is done for parameters from higher levels or set symbols. This means, for example, that if the value of &A is "A,B,C" in "CALL QQSV,(&A)", the second argument will be a sublist. Note that this change can cause problems if the value substituted in a macro call contains unbalanced primes or parentheses.

(7) Symbols appearing in V-type constants appear in the cross-reference listing. They are identified and distinguished from the same symbol defined in this program, if any, by giving VCON as the point of definition.

(8) A C-type constant may have a null value, i.e., "DC C''" is valid and generates nothing. This may help in certain macro definitions.

(9) A type A, Y, S, or Q constant appearing in a DSECT or COM section may have an undefined symbol without being flagged as an error.

(10) A line that is entirely blank except for a possible label is treated as an ANOP pseudo-op.

(11) In open code, an ANOP (or blank line treated as an ANOP) may have a label which is an ordinary symbol and will be defined as the current location counter (or absolute zero if no section has been started). This will not necessarily be aligned to a halfword boundary. An ordinary symbol on an ANOP statement in a macro is still an error.

(12) A length attribute may be used with a literal or set symbol as in " LA 0,L'&ABC" or " LA 0,L'=X'0123'". This will make it unnecessary to use a SETC symbol with a value or one prime in situations of this sort.

(13) An OPSYN or OPDEF statement which attempts to remove an op-code that is already undefined is not an error.

(14) The T', L', S', and I' attributes may be used with a SETC symbol or macro parameter, the value of which is a literal. This means, for example, that a macro can determine the length of a literal passed as a parameter. Note that a reference to a literal in a macro statement or EQU will cause that literal to be generated in the next literal pool, even if it is not otherwise referenced. This may cause the anomaly of an unreferenced literal.

(15) The use of SETC symbols and macro parameters in macro arithmetic expressions has been generalized. The value of the SETC symbol or macro parameter may be any assembler expression that evaluates to an absolute number. All symbols must be predefined and if the difference between two relocatable symbols is used, there must be no location counter discontinuity between them. Note that the expressions must be assembler expressions, not macro expressions. In particular, no ampersands may be used. This change means that the message IEV102 will not occur with EXTEN enabled.

(16) The D' attribute has been extended to allow a macro to determine if a given SETC symbol or macro parameter may be used in a macro arithmetic expression. Use of the D' attribute never produces a diagnostic no matter how erroneous the argument is, and it returns one of four values:

0 The argument is a valid symbol that is not defined.
1 The operand is a symbol or expression that evaluates to an absolute value, i.e., it can be used in a macro arithmetic expression.
2 The operand is a valid symbol or expression, but the value is either relocatable or unknown at this time.
3 The operand has invalid syntax.

(17) The name of a created set symbol may contain any character. This makes created set symbols more useful for building tables which can be searched quickly. This change means that the message IEV083 will not occur with EXTEN enabled.

(18) A Q-type address constant may contain any valid expression. Any references to a DXD or a symbol of a DSECT in a Q-adcon will cause the DXD or DSECT to be placed in the external symbol dictionary (ESD) for the program as a pseudoregister. An A-type address constant may contain references to DXDs or symbols in DSECTs, but only if the DXD or a symbol in the DSECT is also referenced in a Q-adcon. This is because an A-adcon will not cause a DXD or DSECT to be placed in the ESD as a pseudoregister. This is the only difference between A and Q constants. This change means that the message IEV061 will not occur with EXTEN enabled.

(19) The expression on an ORG statement can contain symbols not yet defined in the program. There must be exactly one way in which the locations counters can be assigned, i.e., the ORGs must be resolvable and unambiguous.

(20) The PRINT statement has a new option: "MSOURCE" or "NOMSOURCE". This option is ignored if the GEN option is "NOGEN". Otherwise, it controls whether the source text for macro generated statements is printed as well as the object code. If "MSOURCE" is active (the default), the output is as it always has been. If "NOMSOURCE" is active, the output includes the object code for macro generated statements, but not the source text, i.e., the right-hand side of the listing will be blank for macro generated statements.

(21) There are ten new extended-branch opcodes, five for BCR and five for BC. They are as follows:

BCR	BC	Mask
BGTR	BGT	2
BGER	BGE	A
BEQR	BEQ	8
BLER	BLE	C
BLTR	BLT	4

(22) The attributes of a symbol defined on a DXD will be the same as if the symbol were defined on a DS with the same operands. If NOEXTEN is specified, the attributes will be as if the symbol were a section name.

(23) An attribute reference other than T' to a symbol with type "M" which is otherwise undefined will cause a forward scan for the symbol rather than an error message. A symbol will be type "M" and undefined if it has appeared in the label field of a macro instruction and has not been defined elsewhere.

(24) Underscore ("_") is a legal alphabetic character which can be used anywhere A-Z, #, @, or $ can be used.

(25) Qualified USINGS (or, more properly, labeled USINGS and qualified symbols), allow much greater control over the resolution of symbolic expressions into base-displacement form with specific base registers.

The mechanics of this facility are as follows. First, put a label on a USING statement. Then, to force the assembler to resolve a symbol into base-displacement form through that USING, qualify the symbol by preceding it with the label on the USING, followed by a period. An example of labeled USINGs and qualified symbols would be:

```
     PRIOR USING IHADCB,R10
     NEXT  USING IHADCB,R2
           MVC   PRIOR.DCBLRECL,NEXT.DCBLRECL
```

Without labeled USINGs, the equivalent form would be one of

```
           USING IHADCB,R10
           MVC   DCBLRECL,DCBLRECL-IHADCB(R2)
```

or

```
           MVC   DCBLRECL-IHADCB(,R10),DCBLRECL-IHADCB(R2)
```

The label on a USING may appear in the name field of another statement. The two uses of the symbol are distinct. If a label appears on a USING, any previous USING with the same label is dropped. Labels on USINGs and qualifiers on symbols will appear in the XREF listing flagged by "QUAL" in the definition column.

If qualified symbols are used in an expression, the qualifiers will cancel if possible. The result after all cancellation must be an expression with either no qualifier or one positive qualifier. For example,

```
     THIS.SYM1+NEXT.SYM2-NEXT.DSECT
```

is legal and is an expression qualified by "THIS." On the other hand,

```
     THIS.SYM1+NEXT.SYM2
```

is illegal since neither qualifier cancels. A qualifier may not be used with an absolute symbol and qualifiers may only be used in expressions that will be decoded into a base-displacement address, i.e., in machine instructions or S-type constants, and in USING and DROP statements (see item 26 below).

As is the case with unlabeled USING statements, a symbol (in the first operand) or a register (in any of the remaining operands) may appear in any number of USINGs. However, in the case of qualified USINGs, as long as all the USINGs have unique labels, all are considered active and are eligible to be used as qualifiers.

There is a very basic concept about labeled and unlabeled USINGS that needs to be understood. In non-labeled USINGS, a register implies data, in the sense that a register may imply only one piece of data at a time (i.e., when a register that appeared in a USING appears in another USING, the prior USING is dropped). In labeled USINGs the reverse is true: the data implies a register. That is, a single register may appear in multiple USINGs, all being active, so long as all the USINGs have unique labels. (Dropping of labeled USINGs occurs only when the same

label appears in a USING or DROP statement, not when a repeated register appears.)

Labeled USINGs do not interfere with unlabeled USINGs. When the assembler resolves an implied address into base-displacement form, either the expression to be resolved was qualified or not. If it was qualified, the specified labeled USING will be used. If not, the active unlabeled USINGs will be scanned in the standard manner, looking for a resolution.

A label on a USING defines an environment. As such, to delete that environment, the environment name (the USING's label) must be dropped. An attempt to drop a labeled USING by dropping its registers will result in those registers being dropped instead from the unlabeled USING pool. The labeled USINGs in the example above may be dropped by writing:

```
DROP  PRIOR,NEXT
```

If a symbol in a DROP has been used both as a label on a USING and as an ordinary symbol (so that it could be a register name), the labeled using will be dropped, not the register.

(26) The base specified in a USING statement can now be a simply relocatable expression as well as a register number. This allows one to define nested data structures in the assembler, with the nesting being defined by USING statements. For example if ADATA is a symbol defined in the dsect CMDAREA and UCLOGREC is another DSECT, then one can say

```
USING UCLOGREC,ADATA
```

to specify that the dsect UCLOGREC is to be used to describe the part of CMDAREA starting at ADATA.

USING statements with relocatable bases may be either labeled or unlabeled. Also the relocatable base may be either qualified or unqualified.

As with USING statements specifying registers as bases, there may not be more than one unlabeled USING statement active at the same time for the same relocatable base. Two USING statements that have the same base except that they use different qualifiers are considered to have different bases. A subsequent one for the same base (including qualifier) replaces the previous one. To DROP an unlabeled USING with a relocatable base, specify the base with the appropriate qualifier in a DROP statement (just as for a register USING). Of course, there may be several labeled USINGs active for the same relocatable base at the same time, and one must qualify references to symbols to be resolved through them the same way as if they specified registers for bases.

The rules for deciding which of several relevant USINGs apply to a given address resolution are the same as before: use the smallest displacement, or for equal displacements, the largest register. However, now the active USINGs can define a tree structure and the assembler will follow all paths that lead to register USINGs and pick the best one according to the rules above. There is an arbitrary limit of 25 on the number of different relocatable USINGs that will be considered in resolving one address.

Any one USING statement must contain either exactly one relocatable base or one or more registers. Relocatable bases and registers cannot be mixed in the same statement.

(27) The use of the length attribute of a symbol defined with a DC or DS with the explicit length given by an expression is valid. This is true regardless of whether EXTEN is specified.

I/O UNIT USAGE

The following gives additional details on the logical I/O units used by the assembler.

SCARDS Input

A source programs reads from SCARDS consists of assembler language statements. Each statement is limited to a length of 80 characters. The program is terminated by a statement containing the assembler instruction END.

SCARDS can provide a sequence of one or more assemblies terminated by an end-of-file. When the default option NOBATCH is in effect, the assembler terminates after assembling one source program. When the option BATCH is in effect, the assembler terminates after encountering an end-of-file. (See the earlier section, "BATCH Option," for details.)

Macro Libraries and COPY Sections

Macro libraries may be used by attaching them to logical I/O units 0, 2 through 10. This allows several macro libraries to be used at once. If 0 is not assigned, *SYSMAC will be used. To have no macro library, specify "0=*DUMMY*". If a particular macro name is defined in more than one of the libraries attached, the ordering to establish precedence is the following: 2 through 10, and 0. That is, the assembler searches

the directories of the macro libraries attached to the logical I/O units in that order, and it will use the first occurrence of a macro definition that it finds. Macro definitions included on SCARDS take precedence over all macro libraries. See the sections, "Using Macro Libraries" and "Constructing a Macro Library," later in this volume for details.

| Macro libraries attached to logical I/O units 0 and 2 through 10 may consist of concatenations of line files. Previous versions of the assembler did not allow concatenation on these units and permitted only a single macro library to be assigned to each unit.
|
| Both explicit and implicit concatenation may be used. In the case of implicit concatenation, only $CONTINUE WITH lines appearing in the directory portion of the library will be recognized. $CONTINUE WITH lines should include the RETURN option, otherwise the portion of the directory (if any) following the line will be lost. For consistency and ease of maintenance, implicit concatenation should be restricted to files containing only $CONTINUE WITH lines, rather than embedding such lines in the directory of a macro library. A file containing only $CONTINUE WITH ... RETURN lines referencing other macro libraries is called a "pointer file". Pointer files may point to other pointer files, and this hierarchy may be continued to any depth. Eventually, all pointers must terminate at an actual macro library.
|
| Macro libraries are searched in the order 2 to 10, then 0. In the case of a single macro library attached to each logical I/O unit, the assembler behaves as described above. When concatenation is used, macro libraries comprising the concatenation are searched logically in the same order that would be observed by sequentially reading the directories of the concatenation. Therefore, the precedence of resolution is to the first logical I/O unit on which the name is encountered and within a single concatenation, the first member in which the name is encountered.
|
| Logical I/O unit 0 receives special treatment. If it is left unassigned, the assembler will use *SYSMAC by default. If it desired that *SYSMAC not be searched, and no macro libraries are to be attached to it, unit 0 should be assigned to *DUMMY*. The assembler lists the implicit use of *SYSMAC in the I/O unit summary on the listing head sheet. When unit 0 is assigned to a macro library or libraries and it is desired that *SYSMAC be searched in the normal order, *SYSMAC should be explicitly concatenated to the end of the unit 0 assignment.
|
| Concatenation may be used on an overriding macro library I/O unit supplied when the assembler is invoked as a subroutine.
|
| All macro libraries used will be listed on the head sheet I/O unit summary. In the case of a concatenation, the libraries will be listed successively following the I/O unit to which they are attached. A library will be displayed only once per concatenation, regardless of the number of times it is encountered. Any file of a concatenation which consists only of $CONTINUE WITHs to other files (a pointer file) will

| have the parenthetical comment "no members" printed beside it. The | ordering of the macro library summary from top to bottom is the same as | the order in which the libraries are searched by the assembler.

The COPY assembler instruction may be used to include text from an arbitrary source. In MTS, the operand is not restricted as indicated in the assembler language manual, but may be a full FDname, including line number ranges, explicit concatenation, etc. The total length of the operand may not exceed 63. The operand of the COPY statement is first looked up in the macro directories, in the order specified in the preceding paragraph. If it is not found in any of the macro libraries specified, then it is assumed to be an FDname and the assembler reads from the file or device specified until it encounters an end-of-file. If it is found in a macro library, the assembler reads from the specified place until an end-of-file is sensed. In this latter case, a "$ENDFILE" line will be recognized as an end-of-file even if the system ENDFILE switch has been set OFF (see the SET command description in MTS Volume 1, The Michigan Terminal System, for details).

SPRINT Output

Assembler listing output consists of a heading page, an external symbol dictionary listing, a source and object program listing, a relocation dictionary listing, a symbol cross-reference table, and a diagnostic cross-reference table. Which, if any, of these are produced on SPRINT depends on the options ESD or NOESD, RLD or NORLD, XREF or NOXREF. The default is to produce everything but the RLD listing. The LIST/NOLIST option controls the SPRINT output. If LIST is specified, the output appears as specified. This is the default if SPRINT is not defaulted to a terminal. If NOLIST is specified, no SPRINT output appears. This is the default if SPRINT is defaulted to a terminal. In this case SPRINT is not used.

Each page of listing output normally contains up to 55 lines. This can be changed using the LINECNT option.

Listing output is written in fixed-length print line images. Each line on SPRINT contains one 133-character print line image.

The SPRINT output from ASMH should be largely self-explanatory. If the comments below are insufficient, the user should consult the complete description of the output given in the IBM publication, OS Assembler H Programmer's Guide (form SC26-3759).

ASMH always prints a PRINT statement, regardless of the PRINT options in effect before or after the PRINT statement is executed.

Statements containing variable symbols are printed only as they appear before substitution if NOGEN is in effect.

The sequence ID field (columns 73-80) of a statement generated by a macro call contains information about the origin of the statement in the form "nn-xxxxx", where "nn" is the level of the macro call producing the statement and "xxxxx" is either the statement number of the model statement if the macro definition appears in the source program or the first five characters of the name of the macro for a library macro.

All diagnostics are printed in line by ASMH and a list of flagged lines is printed as part of the summary at the end of the listing.

SPUNCH and Unit 1 Output

Assembler object deck output is written in fixed-length, 80-character card images. The last 8 columns of the card image contain deck identification (taken from the label field of the first labeled TITLE card encountered in the source deck) and sequence numbering. The formats of the card images are described in the section "The Dynamic Loader" in MTS Volume 5, System Services. The options TEST or NOTEST specify whether or not a symbol table is to be included in the object deck. The symbol table is used by the MTS Symbolic Debugging System (see MTS Volume 13, The Symbolic Debugging System).

The output written on logical I/O unit 1 is identical to the object deck written on SPUNCH. The options LOAD or NOLOAD specify whether or not an object deck is to be written on logical I/O unit 1; the options DECK and NODECK specify whether or not an object deck is to be written on SPUNCH. The defaults of DECK and NOLOAD cause only one copy of the object deck to be written, and direct it through SPUNCH.

DIAGNOSTICS AND RETURN CODES

Diagnostic messages are written as part of the assembler listing output. An error occurring in unlisted text will force the erroneous statement to be printed unless the value of the FLAG parameter is not zero.

Each diagnostic message has a message number of the form IEVnnn and a message text.

Associated with any error detected or MNOTE produced is a severity code. The return code produced by the assembler is the highest severity error occurring during the assembly.

Return Code	Explanation
0	No error detected.
4	Minor errors detected; successful program execution is probable.
8	Errors detected; unsuccessful program execution is possible.
12	Serious errors detected; unsuccessful program execution is probable.
16	Critical errors detected; normal execution is impossible.

A diagnostic message for a statement generated by a macro definition may be followed by the macro name or model statement number which caused the error and a SET symbol, parameter number, or value string associated with the error. Parameter 10 is the name field and the rest are numbered upward from 11 with keyword parameters first. This information also may appear for an error on a conditional assembly statement in open code. The macro name in this case will be OPENC.

ASSEMBLER H MESSAGES

Several messages produced by ASMH are different in the MTS version from the messages given in Assembler H Version 2 Application Programming: Guide; there are several mistakes in that publication. The differences and corrections are as follows:

Code	Severity	Message
IEV006	8	OPERAND IS IN A DSECT OR DXD. NO RLD GENERATED. This is a new message. The operand of a statement that causes an RLD entry to be generated is in a DSECT or DXD. No RLD entry is generated.
IEV009	12	SYSTEM VARIABLE SYMBOL ILLEGALLY RE-DEFINED. In addition to the symbols listed, &SYSCCID, &SYSSTYP, &SYSSTMT, &SYSLINE, and &SYSNEST cannot be redefined.
IEV010	12	MACRO PARAMETER EXPRESSION TOO LONG. TRUNCATED AT 864 CHARACTERS. The total length of the parameters on a macro call between commas that appear in the source code is greater than 864. This is not the limit on a single

parameter (which is 255) and can only occur through substitution of a SET symbol or higher-level macro parameter into the macro parameter expression.

IEV014 8 IRREDUCIBLE QUALIFIED EXPRESSION.

Symbol qualifiers are used in such a way that they do not cancel to a single, positive qualifier.

IEV015 8 INVALID USE OF A SYMBOL QUALIFIER.

A qualifier is used with a symbol that is not simply relocatable, or in an expression that does not allow qualifiers.

IEV016 8 USING STATEMENTS NESTED TOO DEEPLY (MORE THAN 25 LEVELS).

More than 25 relocatable USINGs apply to a single base-displacement address.

IEV017 0 UNDEFINED KEYWORD PARAM. DEFAULT TO POSITIONAL INCLUDING KW.

This is now a level 0 message.

IEV032 8 RELOCATABLE OR COMPLEX RELOCATABLE VALUE FOUND IN INVALID CONTEXT.

A relocatable or complex-relocatable expression is used where an absolute or simple-relocatable expression is required.

IEV045 4 REGISTER, BASE, OR QUALIFIER NOT PREVIOUSLY USED.

The operand of a DROP statement cannot be found among the currently active labeled or unlabeled USING statements.

IEV056 4 USING RENDERED NULL BY A PRIOR ACTIVE USING.

Given the rules governing base-displacement resolution of implied addresses (choose the register giving the smallest displacement, and the highest-numbered such register), the USING being processed will never be used because a prior active USING specifies the same base and larger register.

IEV066 0 RELOCATABLE Y-TYPE CONSTANT.

This is now a level 0 message.

IEV094 0 SUBSTRING GOES PAST STRING END.

This message will not occur; i.e., this is not an error.

IEV098 8 ATTRIBUTE REFERENCE TO INVALID SYMBOL.

This can occur when the value of a symbol or expression is requested in a macro arithmetic expression if EXTEN is enabled.

IEV113 4 OPERAND FIELD ENDS PREMATURELY. EXPECTED CONTINUATION NOT PRESENT.

An operand field of SETx, LCLx, GBLx, AIF, or AGO ends with a comma (or an ampersand in the last column for LCLx or GBLx) and there is no continuation card. This may be because the text of the continuation card is taken as comments due to starting past column 16.

IEV114 12 INVALID COPY OPERAND.

This message will not occur.

IEV115 12 COPY OPERAND TOO LONG.

This message will occur only if the COPY operand is longer than 63 characters.

IEV181 12 CCW OPERAND VALUE IS OUTSIDE ALLOWABLE RANGE.

This message will not occur if the third operand is a multiple of 2 (not 8).

IEV980, IEV981, IEV982, IEV983, and IEV999 (various system and I/O messages) cannot occur.

INVOKING ASMH FROM A PROGRAM

The assembler can be called from a program, and for each of the internal I/O "usages," an optional replacement logical I/O unit may be specified.

The invocation can be either a standard S-type subroutine call to the entry point ASMH, found in the file *ASMH, or it can be via LINK, specifying the file *ASMH. The module ASMH is not very large, and the other modules are in shared virtual memory, so using a subroutine call does not cause much increase in virtual memory usage. The prototype given here uses the CALL macro:

```
CALL ASMH,(optionlist[,namelist]),VL
```

where

optionlist is the PAR field, set up as it would be from a $RUN command. (I.e., optionlist is the location of a halfword length-of-text immediately followed by the text. The first character of the text should be the first character that would follow the PAR if it were presented in a $RUN command.)

namelist is a list specifying the names of replacement logical I/O units. The first halfword contains the number of bytes in the remainder of the list. This remainder consists of 8-byte fields. Each of these should be either (a) all binary zeros, or (b) a logical I/O unit name, left-justified, and padded with blanks, or (c) a fullword logical I/O unit number in the first 4 bytes and anything in the second 4 bytes, or (d) a FDUB pointer in the first 4 bytes and anything in the second 4 bytes. Binary zeros indicate the use of the standard I/O unit for that "usage." Entries may be omitted for names beyond the last one to be altered. The order of entries is given in the table below.

Position	Usage	Standard Unit
1	LOAD output	1
2	Unused	
3	Unused	
4	Macro libraries	2-10,0
5	Assembler input	SCARDS
6	Assembler listing	SPRINT
7	DECK output	SPUNCH
8	Unused	
9	TERM output	SERCOM

The return code will be the highest severity code from an error message or MNOTE statement.

The following sample program illustrates how to call ASMH as a subroutine. The program reads input from logical I/O unit 17 and writes the TERM output on logical I/O unit 18.

```
ASMHCALL TITLE 'CALL ASMH AS A SUBROUTINE'
ASMHCALL CSECT
         PRINT NOGEN
         REQU  TYPE=BOTH
         ENTER R12,SA=SA
         CALL  ASMH,(PARLIST,DDNAMES),VL
         EXIT
*
SA       DS    18A
PARLIST  DC    Y(EPARLIST-PARLIST-2),C'RENT'
EPARLIST EQU   *
DDNAMES  DC    Y(EDDNAME-DDNAMES-2),4XL8'0',FL4'17,0',3XL8'0',CL8'18'
EDDNAME  EQU   *
         END
```

May 1983

DIFFERENCES BETWEEN *ASMG, *ASMH, AND *ASMT

The following is a list of some of the differences in the language accepted by the three 360/370/470 assemblers available under MTS at the University of Michigan. The list is not complete and users are referred to the appropriate IBM publications and other sections of this volume for detailed descriptions of the assemblers.

Symbol names in *ASMG and *ASMT are limited to 8 characters in length, while in *ASMH symbol names may be up to 63 characters long. External symbols are limited to 8 characters in length by all three assemblers.

*ASMG and *ASMT (with FMT=CARD, the default) allow source statements to be continued onto two continuation cards, giving a total of three input lines per statement. *ASMH allows source statements to be continued onto nine continuation cards, giving a total of ten input lines per statement. *ASMT also allows an alternate input format (FMT=KEYBOARD).

*ASMT and *ASMH represent self-defining terms as 32-bit values, while *ASMG represents self-defining terms as 24-bit values. Decimal self-defining terms may range from 0 to 16,777,215 (2^{24}-1), in *ASMG, from -2,147,489,648 (-2^{31}) to 2,147,489,647 (2^{31}-1) in *ASMH and from 0 to 4,294,967,295 (2^{32}-1) in *ASMT.

*ASMG and *ASMH allow both unary and binary operators in SETA and SETB expressions, while *ASMT supports only binary operators.

*ASMH allows both unary and binary operators in expressions, while *ASMG and *ASMT support only binary operators.

*ASMG allows up to 5 levels of parentheses in expressions during its conditional assembly phase and up to 11 levels during its assembly phase. In *ASMT, this limit is 64 levels, and in *ASMH, there is no limit.

*ASMG allows up to 16 terms in expressions during its conditional assembly phase and up to 25 terms during the assembly phase. In *ASMT the limit is always 16 terms, while *ASMH has no limit.

*ASMT does not support the ACTR, OPSYN, POP, PUSH, and WXTRN assembler instructions. Both *ASMG and *ASMH support these instructions.

*ASMH supports the AREAD, LOCTR, MHELP, and OPDEF assembler instructions, while *ASMG and *ASMT do not.

*ASMT treats the PUNCH and REPRO assembler instructions as comments, while in *ASMG and *ASMH these instructions cause lines to be included with any object module being written.

*ASMT allows the specification of attributes on CSECT, PSECT, and COM statements, while *ASMG and *ASMH do not. In the MTS environment these attributes have no effect on the object modules produced.

*ASMT supports the PSECT assembler instruction, the system variable symbol &SYSPSCT and R-type address constants, while *ASMG and *ASMH do not. In *ASMT, PSECTS are treated as CSECTS. Each *ASMT control section and entry point may also have a PSECT name defined for it. The PSECT name is the same as the symbolic name, but uses lowercase letters and is the R-value for the symbol.

*ASMG and *ASMH allow the assembler instructions CNOP and ORG to be labeled, while *ASMT does not.

*ASMG and *ASMH allow unnamed DSECTs, while *ASMT requires that all DSECTs be labeled.

The label associated with the TITLE statement in an assembly may be up to 8 characters long in *ASMG and *ASMH, but is limited to 4 characters by *ASMT. In *ASMG and *ASMT, only the first TITLE statement may have a label. In *ASMH, only one TITLE statement may have a label, but the requirement that this statement must be the first has been removed. *ASMG uses up to four characters from the label given with the TITLE statement to label the object module produced. *ASMH will use from 1 to 8 characters to label the object module. *ASMT does not normally label the object module.

*ASMT and *ASMH allow macro definitions to be placed anywhere in the source as long as they occur before they are referenced. *ASMG requires all macros to be defined at the beginning of the source program.

*ASMT supports the FULLGEN option for use with the PRINT assembler instruction, while *ASMG and *ASMH do not.

All three assemblers support the extended form of the EQU instruction. *ASMG and *ASMT allow a length from 1 to 65535 to be specified as an absolute integer expression or a 1- or 2-byte self-defining term, although *ASMG makes the further requirement that the length must be a self-defining term if it is to be used during macro expansion. *ASMH allows a length from 0 to 65535 to be specified as an absolute expression. *ASMH and *ASMT allow the type to be specified as either an absolute integer expression in the range 0 to 255 or as a one-byte self-defining term. *ASMG allows the type to be specified as a one-byte self-defining term, but not as an expression.

*ASMH and *ASMT allow modifier expressions to be used in literals (e.g., =CL(L'SYM)'#'), while *ASMG does not.

*ASMH and *ASMT allow S- and Q-type address constants to be used in literals, while *ASMG does not.

*ASMH and *ASMT do not include the relocatable symbol in a DXD statement in the external symbol dictionary (ESD) unless the symbol is used in a Q-type address constant. *ASMG always includes the symbol in the ESD.

*ASMG and *ASMH allow SETC values up to 255 characters in length, while SETC values are limited to 8 characters in length by *ASMT. The default length for SETC symbols declared without an explicit length on the GBLC or LCLC statements in *ASMG is determined by the PAR field option LSETC which defaults to 8. With *ASMH, no explicit declaration is required, but for compatibility with *ASMG explicit length declarations are accepted and ignored.

*ASMG and *ASMH allow the count (K') attribute to be used with both SETC symbols and macro operands, while in *ASMT the count attribute may only be used with macro operands.

*ASMH allows the count (K') attribute to be used with SETA and SETB symbols in addition to SETC symbols; *ASMG and *ASMT do not.

*ASMH supports the defined attribute (D'); *ASMG and *ASMT do not.

*ASMH allows the number attribute (N') to be used with SETx variables to determine the highest subscript that has been used in an assignment, while *ASMG and *ASMT do not.

*ASMH allows the use of the T', L', S' and I' attributes with SETC symbols, while *ASMG and *ASMT do not.

*ASMT rescans statements after set symbols are inserted in the source text, while *ASMG and *ASMH do not. This rescanning requires that all quotes (') or ampersands (&) must be explicitly doubled if the rescanning is to work without producing error comments. *ASMH rescans macro call operands only and not for ampersands (&).

*ASMH allows the MNOTE assembler instruction to be used in open code as well as within macro definitions. In *ASMG and *ASMT, the MNOTE instruction may only be used from within macro definitions.

*ASMT will accept SETC symbols of the from '123 ' or ' 123' as arithmetic, while *ASMG and *ASMH will not.

In *ASMG and *ASMH when using operand-sublist notation to refer to a macro operand that is not a sublist, the operand itself is returned for the first sublist member and a null value is returned for all other sublist members. In *ASMT, the operand itself is

always returned regardless of which sublist member is being referenced.

The form of the date returned by *ASMH in the system variable symbol &SYSDATE is mm/dd/yy (e.g., 12/31/78), while *ASMG returns yymondd (e.g., 78DEC31), and *ASMT returns mm-dd-yy (e.g., 12-31-78).

The form of the time returned by *ASMG and *ASMT in the system variable symbol &SYSTIME is hh:mm:ss (e.g., 12:00:00), while *ASMH returns the time as hh.mm (e.g., 12.00).

*ASMH allows COPY instructions to be nested within copied code to an unlimited depth, *ASMG allows nested COPY instructions to a depth of 5 and *ASMT does not allow nested COPY instructions.

*ASMG and *ASMT do not allow the assembler instructions END, ISEQ, MACRO, MEND, or OPSYN (in fact *ASMT doesn't allow OPSYN anywhere) to occur within macro definitions, while *ASMH does. In addition, *ASMT does not allow the START assembler instruction to occur within macro definitions.

*ASMT and *ASMH both allow the assembler instructions END and ISEQ to appear within copied code, while *ASMG does not.

*ASMH and *ASMT allow the assembler instructions CSECT, DSECT, END, MNOTE, OPSYN (*ASMH only), PRINT, and START to be generated using variable symbols, while *ASMG does not.

*ASMT allows the assembler instructions AGO, AIF, ANOP, GBLx, ISEQ, LCLx, REPRO, and SETx to be generated using variable symbols, while *ASMG and *ASMH do not.

*ASMH allows nested macro definitions, while *ASMG and *ASMT do not.

*ASMH allows macro definitions to be redefined, while *ASMG does not. *ASMT also allows macro definitions to be redefined, but unlike *ASMH, the macro definition that occurs physically last in the source deck will be used for all references to the macro.

*ASMG does not allow variable symbols in the label or operand fields of the COPY, ISEQ, REPRO, or OPSYN assembler instructions. *ASMH and *ASMT do not have this restriction.

*ASMH allows multilevel sublists in macro calls, while *ASMG and *ASMT do not.

*ASMT and *ASMH allow comments (both * and .*) to occur before the MACRO statement in macro library files, *ASMG allows only regular comments (*) before the MACRO statement.

ASMT allows macro comments (.) to occur in open code as well as within macro definitions, while *ASMH and *ASMG do not.

*ASMH allows comments (both * and .*) between the MACRO statement and the macro prototype, while *ASMG and *ASMT do not.

*ASMH relaxes the ordering requirements for mixed-mode format macros allowing keyword and positional parameters to be intermixed. *ASMG and *ASMT require all positional parameters to occur before the first keyword parameter.

*ASMH supports extended forms of the AGO, AIF, GBLx, LCLx, SETA, SETB, and SETC assembler instructions, while *ASMG and *ASMT do not.

*ASMH allows SET symbols to be declared explicitly using the GBLx or LCLx assembler instructions or implicitly by appearing in the name field of a SETx statement. These declarations can appear anywhere so long as they appear before the symbol is first used.

*ASMH and *ASMT allow more than one GBLx or LCLx assembler instruction for a given set symbol to appear as long as only one is encountered during macro expansion (through the appropriate use of AIF and AGO instructions).

*ASMH and *ASMT allow macro calls to be generated by substitution, while *ASMG does not.

*ASMH and *ASMT allow macros to redefine machine instruction mnemonics, although *ASMT will produce a warning comment. *ASMG does not allow machine instruction mnemonics to be redefined in this manner.

*ASMH allows macros to redefine assembler instructions, while *ASMG and *ASMT do not.

*ASMH allows embedded equal signs (=) to appear in positional macro parameters, although a warning message will be produced. *ASMG and *ASMT do not allow embedded equal signs to appear in positional macro parameters.

*ASMH allows SETA expressions where SETB expressions are legal, with a value of zero taken as false and nonzero values taken as true. *ASMG and *ASMT do not allow SETA expressions in place of SETB expressions.

*ASMH allows SET symbols to be created by substitution, while *ASMG and *ASMT do not; *ASMH allows created SET symbols for this purpose.

*ASMH allows duplication factors expressed as SETA expressions enclosed in parentheses to be used with SETC symbols and character strings in SETC expressions.

In *ASMH, literals can be used in EQU statements such as "A EQU =A(ABC)" and in expressions such as "IC 0,=X'010203'-1". *ASMG and *ASMT do not allow this.

In *ASMH, a predefined, absolute expression can be used in a SETA or SETB expression and as the value of a SETC symbol or macro parameter used in a SETA or SETB expression. For example, "&A SETA JTBLPOOL+2" or "AIF (LENGTH GT 256).ERROR". The symbols used must be predefined and will not be cross-referenced. *ASMG and *ASMT do not allow this. *ASMT allows an expression containing SDTs or variable symbols but not ordinary symbols.

In *ASMH, the value of &SYSNDX is 5 instead of 4 digits long. This will not cause a problem with symbols that are too long since Assembler H allows symbols up to 63 characters long.

In *ASMH, macro call operands are scanned for sublists, keywords, etc., after all substitution is done for parameters from higher levels or set symbols. This means, for example, that if the value of &A is "A,B,C" in "CALL QQSV,(&A)", the second argument will be a sublist. Note that this change can cause problems if the value substituted in a macro call contains unbalanced primes or parentheses.

In *ASMH, a C-type constant may have a null value, i.e., "DC C''" is valid and generates nothing. This may help in certain macro definitions. *ASMG and *ASMT do not allow this.

In *ASMH, a type A, Y, S, or Q constant appearing in a DSECT or COM section may have an undefined symbol without being flagged as an error. *ASMG and *ASMT flag this as an error.

In *ASMH, a line that is entirely blank except for a possible label is treated as an ANOP pseudo-op.

In open code with *ASMH, an ANOP (or blank line treated as an ANOP) may have a label which will be defined as the current location counter (or absolute zero if no section has been started). This will not necessarily be aligned to a halfword boundary. An ordinary symbol on an ANOP statement in a macro is still an error.

In *ASMH, a length attribute may be used with a literal or a set symbol as in " LA 0,L'&ABC" or " LA 0,L'=X'0123'". This will make it unnecessary to use a SETC symbol with a value or one prime in situations of this sort. In *ASMT, a length attribute may used only with a set symbol; in *ASMG, a length attribute may not be used with either a literal or a set symbol.

In *ASMH, an OPSYN or OPDEF statement which attempts to remove an op-code that is already undefined is not an error (Assembler G treats this as an error while Assembler T does not support either OPSYN or OPDEF).

In *ASMH, expressions on statements that affect the location counter (e.g., ORG, CNOP) and the first operand of an EQU may contain symbols defined later in the assembly.

*ASMH supports the MSOURCE/NOMSOURCE option of the PRINT statement.

The set of extended-branch opcodes recognized is different for each of the three assemblers.

*ASMH defines the attributes of a symbol on a DXD opcode correctly. *ASMG and *ASMT define the attributes as if the symbol was a section name.

*ASMH allows the underscore (_) wherever an alphabetic character is allowed.

*ASMH allows labeled USINGs and qualified symbols. It also allows relocatable base values and registers in USINGs.

The following table indicates which system variable symbols are supported by each of the three assemblers, and whether the symbols may be used in open code as well as macro definitions or just in macro definitions.

Name	*ASMG	*ASMH	*ASMT
&SYSDATE	macros	open code, macros	open code, macros
&SYSECT	macros	macros	macros
&SYSLINE	------	macros	------
&SYSLIST	macros	macros	macros
&SYSLOC	------	macros	------
&SYSNDX	macros	macros	macros
&SYSPARM	macros	open code, macros	------
&SYSPSCT	------	------	macros
&SYSSTYP	macros	macros	macros
&SYSSTMT	------	macros	------
&SYSTIME	macros	open code, macros	open code, macros
&SYSNEST	------	macros	------

May 1983

UTILITY PROGRAMS FOR ASSEMBLER USERS

The following public file descriptions also appear in MTS Volume 2, Public File Descriptions, and are repeated here for the user's convenience.

*ASMTIDY

Contents: The 360/370 assembly "tidying" program.

Purpose: To edit 360/370 assembly programs into an easily readable format and to indent programs containing structured macros to show the program structure.

Use: The program is invoked by the $RUN command.

Program Key: *ASMTIDY

Logical I/O Units Referenced:

SCARDS - the input file consisting of an untidied assembly source.
SPUNCH - the output file to contain the tidied assembly source.
SPRINT - the listing of the tidied source plus MTS line numbers of SPUNCH, statement numbers, and level numbers.
SERCOM - severe error comments.

Parameters: The PAR field of the $RUN command can be used to change the assembly standard format. The form is PAR=N,O,V,C where:

N defines the starting column for the name field
O defines the starting column for the operation field
V defines the starting column for the operand field
C defines the starting column for the comments field

The default is PAR=1,10,16,35. Only those parameters whose values differ from the default need be specified. "Missing" parameters may be represented by a single comma or an explicit zero.

Example: PAR=,8,,30 will change the operation field and comments field starting in columns 8 and 30, respectively, but will not alter the others.

The following parameters may be specified, after "N,O,V,C" if any, in the parameter field of the $RUN command. The parameters must appear after "N,O,V,C" and must be separated by commas or blanks. In case of conflicting parameters, the rightmost parameter takes precedence. Some of the parameters, as indicated below, may be negated by prefixing them with "NO", "N", "-", or "¬". Alternatively, these same parameters may be written as

parm=ON, parm=YES, parm=NO, or parm=OFF, where "parm" is the parameter name. Thus, LIST is the same as LIST=ON, and NOLIST the same as LIST=OFF. No embedded blanks are allowed within a parameter.

[NO]BATCH The BATCH parameter specifies that *ASMTIDY is to process a stream of assemblies, with the last assembly terminated by an end-of-file. The NOBATCH parameter causes *ASMTIDY to process only one assembly. The default is BATCH.

[NO]DECK The DECK parameter specifies that tidied lines are to be written on SPUNCH. The NODECK parameter suppresses the SPUNCH output. The default is DECK.

[NO]EDIT The EDIT parameter specifies that *ASMTIDY is to edit all machine instructions plus two assembler instructions USING and DROP. General registers 0...15 are replaced by R0...R15 whenever possible; floating registers 0...6 replaced by FR0...FR6; and control registers 0...15 by CR0...CR15. If any suboperand of an assembler operand is omitted, e.g., BASEOF(, 15), a zero is inserted in its place. If D2(X2,B2) or S2(B2) is written like D2(0,0) or S2(0) respectively, *ASMTIDY will edit as D2 or S2 respectively. The instructions, BC and BCR, are replaced by extended branch instructions whenever possible. The default is NOEDIT.

[NO]FRAME The FRAME parameter specifies that the comment lines preceded by "*FRAME" are to be enclosed in a box of asterisks. If the seventh character of the *FRAME line is nonblank, it is taken as the framing character instead of the asterisk. The NOFRAME suppresses this feature. The default is FRAME.

INDENT=n The INDENT parameter specifies the number of spaces used to indent the source statement for each nesting level. The default indentation is 2 spaces per nesting level. This parameters applies to the source files containing structured programming macros (IF, ELSE, DO, etc.). To disable the INDENT feature, the user may specify INDENT=0 or INDENT=OFF.

INPUT={CARD|FREE}
The CARD parameter means the input consists of 80-character cards. A warning will be printed for every line of length more than 256 characters and for every line with any nonblank

character beyond column 80. The FREE parameter specifies the free-format input. Each input line may be up to 256 characters. Continuation is indicated by a minus sign (-) as the last character of the line. The input parameters INPUT=CARD and INPUT=FREE may be abbreviated to CARD and FREE respectively. The default is CARD.

[NO]LCTOUC
The LCTOUC parameter specifies that assembler labels, opcodes, and operands are to be translated to upper case, except those within quotes. This parameter is useful if the source file was typed in lower case. The comment fields remain unchanged. The default is LCTOUC.

LINECNT=n LINECNT specifies the number of lines per page to be printed. The range is 5 to 32767; the default is 60.

[NO]LIST The LIST parameter causes an edited listing of the source program to be produced on SPRINT. The line numbers printed are those used on SPUNCH. NOLIST suppresses this listing. The default is LIST unless SPRINT defaults to a terminal.

MAXLEN=n If the parameter OUTPUT=FREE is in effect, the parameter MAXLEN specifies the maximum output length of lines on SPUNCH. "n" must be >= 72 and <= 256. The default is MAXLEN=80.

OUTPUT={CARD|FREE}
This is same as INPUT parameter except this goes for the SPUNCH output. The default is OUTPUT=CARD. For OUTPUT=FREE, see also the parameter MAXLEN.

[NO]SEQ The SEQ parameter specifies that *ASMTIDY is to punch the sequence ID field in columns 73-80 of the punch output. Columns 73-76 contain the first four characters in the label of the TITLE statement, and columns 77-80 contain the sequence number. This action will be overridden if the ending column as specified by the ICTL statement is greater than 72 or if the ending column is 72 with the continuation column specified in the ICTL statement. The default is SEQ=OFF.

TLC or FULLTLC

The TLC (Translate to Lower Case) parameter specifies that comment fields on assembler instructions are to be translated to lower case. The first character of the comment field is not changed. If the comment is already in mixed case, it is not changed. FULLTLC specifies that, in addition to comment fields, all comment statements (lines beginning with "*") are also to be translated to lower case. By default, no comments are translated.

VERBOSE or TERSE

The VERBOSE parameter specifies that *ASMTIDY is to confirm that an assembly source program has been tidied by issuing the following message

ASSEMBLY SOURCE TIDIED FOR name

where "name" is the label of the TITLE statement in the source program. The TERSE parameter suppresses this message. The default is the same as the setting of the MTS $SET TERSE option.

Description: *ASMTIDY may be used to tidy 360/370 assembler source programs. It reads an untidied assembler source program from the logical I/O unit SCARDS and writes out the tidied source on the unit SPUNCH. If LIST is specified, *ASMTIDY will produce an indented listing of the source program showing any program structure (if structured programming macros are used).

The program will set up each field to standard format with the name field in column 1 (or the beginning column specified by the ICTL statement), the operation field starting in column 10, the operand field starting in column 16, and the comment field in column 35. If the last character of any field exceeds the starting column for next field, the starting column will be one blank after the last character. If structured programming macros are used (IF, ELSE, DO, etc.), each statement beginning with the operation field will be indented 2 spaces for each nesting level unless they are enclosed within the pseudo-operations MACRO and MEND. This spacing can be overridden by the INDENT=OFF parameter. For example, the operation field of each statement following an IF macro at level 0 will begin in column 12, for level 1, the operation code will start in column 14. Comment fields still begin in column 35 if possible; if not, they are pushed to the right. It is possible that the operation code may not be indented since the assembler

requires the operation code to be complete in the first line.

The program will automatically translate label names, operation codes, and operands to upper case except for characters enclosed within single quotes. Comments are left unchanged.

Comment cards ("*" in column 1 or the beginning column specified by the ICTL statement) are not modified, except when "*FRAME" starts a line preceding a series of comment cards, in which case a box comprised of asterisks will be built around those comment cards. This may be suppressed with the NOFRAME parameter.

Examples:

```
$RUN *ASMTIDY SCARDS=A SPUNCH=B
```

In the above example, the source program is read from file A and the tidied output is written to file B. The standard format is used.

```
$RUN *ASMTIDY SCARDS=X SPUNCH=Y PAR=,,20,40,FREE
```

In the above example, the input in the file X is free-formatted. The output in file Y is converted to card-formatted output with the operand field in column 20 and the comment field in column 40.

*PEXIT

Contents: An Assembler-H listing postprocessor.

Purpose: To make cosmetic alterations to the source program listing generated by *ASMH. This will provide a more readable listing, especially if the structured programming macros are used.

Use: The subroutine is invoked by the H-level Assembler by specifying the PEXIT parameter, e.g.,

```
$RUN *ASMH [I/O units] PAR=PEXIT=*PEXIT
```

Program Key: *EXEC

Description: The subroutine is called for every line printed by the assembler to edit the listing as follows:

(1) The current nesting level for the structured programming macros (IF, DO, etc.) is inserted in the output for each statement.
(2) The cross-reference listing is pruned of all the internal labels generated by the structured programming macros (e.g., IF#nnn, DO#nnn, etc.), the MSG macros, and some additional *SYSMAC macros.
(3) CASE macro invocations will have their respective case number printed in the right-hand margin of the listing. If a CASE macro specifies a list of cases, only the first number is printed suffixed by the character "*", indicating that more than one case was given.
(4) FLAGS macro invocations will have the flag mask, name, and address printed in the left-hand margin of the listing for each flag declared. If more than one flag is specified on the same line of a FLAGS macro call, only the first will have this information listed.
(5) Macro invocations will have whatever object code is generated printed on the same line as the macro call. Only the first one or two instructions (up to eight bytes) will be listed by default. If PRINT GEN,NOMSOURCE is in effect, the output will include the object code for all macro generated statements, with the source text blanked.

The first two items do not require any special macro calls in the source file or macro libraries for the print exit subroutine. The additional features, however, re-

quire the macro library *SYSMAC (the standard macro library used by *ASMH), and that the following macro statement be coded in the source file before any USING or DROP statements:

```
ASMEDIT [csect][,keywords]
```

The positional parameter "csect" is optional but should specify the name of the initial CSECT if the macro is coded outside of a control section. The ASMEDIT macro initializes the print exit environment, redefining the following assembler instructions:

```
USING
DROP
PUSH
POP
PRINT
EJECT
TITLE
SPACE
```

ASMEDIT also defines the symbolic register names R0-R15 and emits a USING for each of the registers with null dsects to allow an abbreviated form of address specification using the $ symbol. This permits instruction operands to be written as follows:

```
LA    R2,$R2+1
MVC   $R1(4),=C'QQSV'
```

instead of

```
LA    R2,1(,R2)
MVC   0(4,R1),=C'QQSV'
```

This notation may be used only for registers that do not have another unlabeled USING currently active.

ASMEDIT Macro Options

The ASMEDIT macro accepts a number of "keyword" parameters that may be used to change some of the *PEXIT defaults and request additional postprocessor functions. The following keywords are recognized:

BOX={YES|NO|len|(shift,len)}

The BOX option requests *PEXIT to build *BOX comments if YES; not to if NO; sets the BOX length to "len" (4<=len<=72); shifts the box right or left (if "shift" is negative) on the listing page by "shift" columns. The box cannot be shifted left more than

54 columns (shift>-54) or shifted right such that "shift" + "len" exceeds 80.

INDCH={c|NO}

The INDCH option sets the register indirection prefix character to "c". The default character is $. If INDCH=NO is specified, the register null dsect usings will not be generated so that the register indirection notation will not be available.

LINECOUNT=n

The LINECOUNT option is the same as the LINECOUNT assembler option. The default number of lines between new page headings is 55. This option must be specified in addition to the assembler parameter since *PEXIT repaginates the listing.

LITXREF=YES

The LITXREF option requests *PEXIT to generate a separate literal cross-reference listing that is reformatted like the one produced by *ASMG. If a large number of literal symbols is used, this option may potentially reduce the number of literal cross reference pages by half.

REQU=NO

The REQU option suppresses the generation of the register equates by the ASMEDIT macro. The register indirection usings will still be emitted unless INDCH=NO is also specified.

TOC=YES

The TOC option requests *PEXIT to generate a table of contents at the end of the listing. The table of contents entries are the operands of all nonblank TITLE statements, including TITLEs before this macro call.

Only the first ASMEDIT macro statement encountered will be printed in the listing by *PEXIT.

PEXIT Control Cards

Some *PEXIT options are specified on Assembler comment statements. The following options control the output produced by the listing postprocessor.

*FRAME

The FRAME option builds a box composed of asterisks around the following comment lines. If the seventh character of the *FRAME line is nonblank, it is taken as the framing character, instead of the asterisk. The box width is 71.

*BOX

The BOX options builds a box composed of TN box characters around the following comment lines. If "*BOX CENTER" is specified, the box will be centered horizontally on the listing page. The default box width is 72.

Example:

```
$RUN *ASMH SCARDS=SOU SPUNCH=OBJ SPRINT=*PRINT*
     PAR=TEST,PEXIT=*PEXIT
```

May 1983

MACRO LIBRARIES

This section is concerned with macro libraries that may be used with Assembler G, Assembler H, and the TSS Assembler. The beginning of this section describes how to use and construct macro libraries.

The bulk of this section is composed of descriptions of macros provided by the Computing Center for assembly language programmers. These macros are all in *SYSMAC (the system macro library for *ASMH), in *ASMGSYSMAC (the system macro library for *ASMG), and in *ASMTSYSMAC (the system macro library for *ASMT). Many of these macros aid the programmer in providing calling sequences for subroutines of the same name as the macro. The macro descriptions found in this section assume the user is familiar with the related subroutine descriptions, which are to be found in MTS Volume 3, System Subroutine Descriptions.

USING MACRO LIBRARIES

The Computing Center maintains a number of macro libraries in public files. In addition, users can construct and use their own macro libraries.

Except for the system macro library *SYSMAC, any macro library that is to be used when assembling a program must be explicitly mentioned when running *ASMH. Several macro libraries may be used for one assembly. A macro library is specified by assigning it to one of the logical units 2-5, or 0 when running the assemblers. Logical I/O units 6-10 also may be used with *ASMH to specify additional macro libraries. For example,

```
$RUN *ASMH SCARDS=SOURCEPGM SPUNCH=OBJ 0=MACLIB
```

will use MACLIB as a macro library. For *ASMH, the macro libraries are searched in the order of logical I/O units 2 though 10, followed by I/O unit 0. For example,

```
$RUN *ASMH SCARDS=IN SPUNCH=OBJ 2=MYMACLIB 0=*SYSMAC
```

will cause a macro to be expanded from MYMACLIB if that macro's definition is found there. Otherwise, the definition from *SYSMAC will be used. Note that any macro definitions supplied with the assembler input will take precedence over definitions in a macro library.

The following public files contain macro libraries:

*SYSMAC

*SYSMAC is the system macro library. These macros may be used with the Assembler H (*ASMH). These macros are described below.

*OSMAC

*OSMAC contains the macro library from IBM's Operating System. It is designed to enable the assembling of OS programs under MTS. The programs so assembled must not be run under MTS. Descriptions of these macros will be found in the pertinent IBM documentation.

CONSTRUCTING A MACRO LIBRARY

As described below, a macro library has a rather simple structure. Small macro libraries can be easily constructed by hand. For constructing larger macro libraries, the program *MACUTIL is available (see the section "The Macro-Library Editor" in this volume).

A macro library is a line file containing both a directory of the names and the definitions themselves.

A. The directory:

1. Each entry of the directory contains the name of a macro or copy section starting in column 1 and the line number of the macro definition header of the corresponding macro or the first line of a copy section separated from the name by at least one blank.

2. The line number of the first entry in the directory must be 1.

3. The terminating entry in the directory is a string of eight zeros in columns 1-8.

B. The macros or copy sections:

1. The line number of the macro-definition header of each macro or the first line of a copy section must be a positive integral number.

2. The first macro or copy section follows the last entry in the directory.

3. A copy section should be terminated by $ENDFILE.

Example:

```
$COPY *SOURCE* FILE(1)
BASR      10
BAS       20
00000000
$ENDFILE
$COPY *SOURCE* FILE(10)
          MACRO
&LABEL    BASR    &REG1,&REG2
&LABEL    BALR    &REG1,&REG2
          MEND
$ENDFILE
$COPY *SOURCE* FILE(20)
          MACRO
&LABEL    BAS     &REG1,&LOC
&LABEL    BAL     &REG1,&LOC
          MEND
$ENDFILE
```

The public file *MACUTIL contains a program to construct a macro library. Before the program is run, the macro definitions should be put in the line file starting at some relatively high, positive line number. The MACRO line of each definition must occur on an integral line number. Then *MACUTIL is run to construct the directory, which must start at

line 1 of the file. *MACUTIL reads the macro definitions from the file attached to logical I/O unit 0 and writes the directory back into the same file, i.e.,

```
$RUN *MACUTIL 0=macrofile PAR=BUILDIR
```

The following would produce a macro library from the same macros as the previous section:

```
$COPY *SOURCE* FILE(1000)
          MACRO
&LABEL    BASR    &REG1,&REG2
&LABEL    BALR    &REG2,&REG2
          MEND
          MACRO
&LABEL    BAS     &REG1,&LOC
&LABEL    BAL     &REG1,&LOC
          MEND
$ENDFILE
$RUN *MACUTIL 0=FILE PAR=BUILDIR
```

Further details on the use of *MACUTIL are given in the section "The Macro-Library Editor" in this volume.

SYSTEM-SUPPLIED MACROS

The following pages contain descriptions of the macros available in *SYSMAC to be used with the Assembler H (*ASMH). They are arranged alphabetically by macro name. The structured-programming macros (IF, DO, etc.) are described in the section "Structured Programming Macros." Many of these macros aid the programmer in providing calling sequences for subroutines of the same name as the macro. Descriptions of those macros assume the user is familiar with the related subroutine descriptions, which can be found in MTS Volume 3, System Subroutine Descriptions.

If not otherwise specified by the user, the Assembler H defaults logical I/O unit 0 to *SYSMAC.

The macros described in this section will assemble correctly only with the Assembler H. Both *ASMGSYSMAC and *ASMTSYSMAC contain versions of many of these macros that will correctly assemble with the Assembler G and the TSS Assembler. However, since the macro language of these assemblers is much more limited than that of the Assembler H, not all the macros and features described are available to them. The availability of a macro in *ASMGSYSMAC or *ASMTSYSMAC may be ascertained by directly examining the macro library directory at the head of the macro file.

May 1983

ASMTYPE

Macro Description

Purpose: To determine which assembler is being used to assemble the program.

Prototype: [label] ASMTYPE

Description: The global SETC symbol &SYSASM is set to 'G', 'H', or 'T' to indicate whether ASMG, ASMH, or ASMT is being used to assemble the program. &SYSASM must be declared with a GBLC op-code in order to be properly used.

A8, S8, A8R, S8R

Macro Description

Purpose: To add or subtract 8-byte integers.

Prototype:
```
[label]  A8    r1,loc[,F1=f1][,CC=Y]
[label]  S8    r1,loc[,CC=Y]
[label]  A8R   r1,r2[,F1=f1][,CC=Y]
[label]  S8R   r1,r2[,CC=Y]
```

Parameters:

r1 is an even general register

loc is an 8-byte storage area on a fullword boundary

r2 is an even general register

f1 is the location of a fullword constant one (if this parameter is omitted, the literal =F'1' will be used instead)

Description: The 8-byte integer in r2 and r2+1 or in loc through loc+7 will be added to (for A8 and A8R) or subtracted from (for S8 and S8R) the 8-byte integer in r1 and r1+1. The result will be placed in r1 and r1+1.

An 8-byte integer has the same twos-complement form as a 2- or 4-byte integer, but is 8 bytes (64 bits) long.

If CC=Y is specified, the condition code will be set as after an A or S instruction. Otherwise, the condition code will be unpredictable.

Warning: The code generated by the A8 and A8R macros may cause a fixed-point overflow exception in cases where the values are very large (close to overflowing a double register) even though the final result would be between -2^{63} and $2^{63}-1$. Similarly, the code generated by the S8 and S8R macros may fail to cause a fixed-point overflow exception when logically it should cause an exception. In the cases where an overflow does occur, the condition code will be 0, 1, or 2, not 3 as would normally occur with the A and S instructions.

Examples:
```
LAB1     A8    2,X
```

This example adds the 8 bytes at X to the number in registers 2 and 3.

```
LAB2     S8R   2,6,CC=Y
```

This example subtracts the number in registers 6 and 7 from the number in registers 2 and 3. The condition code is set depending on the results.

```
LAB3     A8    2,Y,F1=ONE
```

The number at location Y is added to the number in registers 2 and 3. The constant 1 from location ONE is used instead of the literal =F'1'.

ASSIGN

Macro Description

Purpose: To generate tables of constants.

Prototype: [label] ASSIGN mode,values

Parameters:

mode is any legal assembly-type character for a DC statement (e.g., F, E, D, A, etc.).

values is a parenthesized list of values to be assigned. For each element in this list, a DC statement is generated using mode as the type and the element as the value.

Example:

```
TBL   ASSIGN  E,(1.0,1.1,3.0,4.0)
```

This example generates a table of the form

```
TBL   DC    E'1.0'
      DC    E'1.1'
      DC    E'3.0'
      DC    E'4.0'
```

May 1983

BPI

Macro Description

Purpose: To branch on a program interrupt.

Prototype: [label] BPI type,loc

[label] BPI (type1,type2,...),loc

Parameters:

type,type1,type2
: is the category or categories of program interrupts to be covered by the macro. See the description below for the type codes available.

loc
: is the location to transfer to if the specified type (or types) of program interrupt occurred on the previous instruction.

Description: Two forms of the macro are available: the first form is used to specify one branch address for the corresponding category of interrupt type; the second form is used to specify one branch address for several different interrupt-type categories. Several BPI macros may be given in succession.

When an instruction gets a precise program interrupt, the following instruction is checked to determine if it is a BPI instruction. If it is, the type of program interrupt that occurred is compared with the type categories specified in the BPI macro. If there is a match, the condition code is set to reflect the interrupt that occurred (according to the table below) and the branch is taken. Otherwise, the next instruction is checked to determine if it is a BPI instruction, etc. If there is no BPI transfer made (either because there was no BPI instruction or because the program interrupt type did not match the mask of any BPIs that were present), then the normal processing of the interrupt occurs. Namely, if a PGNTTRP exit is active, it is taken; otherwise, an error comment is printed and a return is made to MTS command mode.

Table of BPI interrupt-type categories:

BPI Type	BPI Mask	Int. Number	Interrupt Name	Condition Code on Branch
OPCD	8	1	Operation	1
		2	Privileged operation	2
		3	Execute	3
OPND	4	4	Protection	0
		5	Addressing	1
		6	Specification	2
		7	Data	3
OVDIV	2	8	Fixed overflow	0
		9	Fixed divide	1
		10	Decimal overflow	2
		11	Decimal divide	3
FP	1	12	Exponent overflow	0
		13	Exponent underflow	1
		14	Significance	2
		15	Floating-point divide	3

Precise program interrupts: For a user program running under MTS on an IBM 360/67, only program interrupt 4 (protection) is potentially imprecise. For program interrupt 4, a fetch-protect violation is always precise. A store-protect violation is precise only if

(1) the CPU must also wait for the operation to fetch something (hence, the TS instruction causes a precise interrupt), or
(2) more than one doubleword of storage must be changed:

a. STM of 1 register - imprecise
   STM of 2 registers - imprecise, if storage specified is doubleword-aligned; precise, otherwise.
   STM of 3 or more registers - precise
b. variable-length instructions - precise if destination operand crosses a doubleword boundary.

All program interrupts on any 370 are precise.

This macro assembles as a special type of NOP instruction; hence, if executed, it is treated as a "branch never" instruction. The form of the BPI instruction is

470xbddd

where x is the BPI mask as given in the table above, and b and ddd are the base and displacements specifying the branch address.

Examples:

```
SUBR  STM   14,12,12(13)
      BPI   OPND,BADR13
```

This example will branch to the address BADR13 on an interrupt caused by an illegal value in register 13.

```
      D     0,0(3)
      BPI   OVDIV,BADDIV
      BPI   OPND,BADREG3
```

This example will branch to the address BADDIV on an interrupt caused by a fixed-divide exception, or to the address BADREG3 on an interrupt caused by an illegal value in register 3.

```
      EX    5,INST
      BPI   (OPCD,OVDIV,FP),BAD
```

This example will branch to the address BAD on any interrupt covered by the OPCD, OVDIV, or FP categories.

CALL

Macro Description

Purpose: To pass control to a subroutine at a specified entry point.

Prototype: [label] CALL name[,(par)[,VL]][,ID=num] [,MF=mod] [,EXIT=exitseq][,LIT=YES]

Parameters:

name is the name of the entry point to be given control; the name is used in the macro instruction as the operand of a V-type address constant. If (15) is specified, GR15 must contain the address of the entry point to be given control.

par (optional) are one or more address parameters, separated by commas, to be passed to the subroutine. Each address is expanded, in the order specified, to a fullword on a fullword boundary. When control is passed to the subroutine, GR1 contains the address of the first parameter. If no address parameters are specified, the contents of GR1 are not changed. The address parameter may be a general register number enclosed in parentheses, in which case the value in the specified register will be inserted into the parameter list pointed to by GR1. The address parameter may also be a literal unless the MF=L parameter is being used and the program is being assembled by ASMG or ASMT (if ASMH is being used, the literal is always legal). In this case, the necessary instructions are generated to load the address of the literal and store it in the parameter list (if ASMH is being used, the literal will be assembled directly into the parameter list).

VL (optional) specifies a variable number of parameters being passed to the subroutine. VL causes the high-order bit of the last parameter in the macro expansion to be set to 1; the bit can be checked by the subroutine to find the end of the parameter list. If VL is specified when par and mod are omitted, then GR1 will be set to zero before the call.

num (optional) is a keyword parameter containing a number not exceeding 2**16-1 (65535). The last fullword of the macro expansion is a NOP instruction containing num in the low-order two bytes. GR14 contains the address of the NOP instruction when the subroutine is given control. If num is omitted, a NOP is not generated.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

LIT=YES (optional) specifies that literals are addressable and the macro will then use a literal for the entry address rather than generating an adcon inline. If LIT is not specified, then literals will not be used unless a LITADDR or MACSET macro has specified otherwise.

This macro destroys the contents of registers 14 and 15 (and, if any of par, mod, or VL is given, the contents of register 1) in setting up the call. Register 0 may be used to return a value from the called subroutine. Register 13 must point to the calling program's save area. The called subroutine may set the condition code.

Description: The linkage relationship established when control is passed to the subroutine is the same as that created by a BAL instruction; that is, the calling program expects control to be returned. The calling program is not involved in passing control, so the reusability status of the subroutine must be maintained by the user.

If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced and all other parameters are optional.

If MF=(E,listadr) is specified, listadr is assumed to be the address of a remote parameter list, and only the executable code required to call the subroutine is generated. The address of the parameter list can be

coded as described under par, or can be loaded into GR1, in which case MF=(E,(1)) should be coded.

Examples:

```
      LAB1  CALL  SYSTEM
```

This example assembles a call to the SYSTEM subroutine.

```
      LAB2  CALL  SCARDS,(REG,LEN,MOD,LNUM)
```

This example assembles a call to the SCARDS subroutine. Register 1 points to the parameter list containing the addresses of REG, LEN, MOD, and LNUM.

```
      LAB3  CALL  (15),(PAR1,PAR2)
```

This example assembles a call to the subroutine whose address is contained in register 15. Register 1 points to the parameter list containing the addresses of PAR1 and PAR2.

```
            CALL  GETFD,MF=(E,FNAME)
            .
            .
      FNAME DC    C'DATAFILE '
```

This example assembles a call to the GETFD subroutine. The address of the parameter list FNAME is stored in register 1.

```
            L     7,TXTPTR
            CALL  KEYWORD,(LHTLEN,LHTAB,EXTAB,(7),RHTAB)
```

This example assembles a call to the KEYWORD subroutine. Register 1 points to the parameter list containing the address of LHTLEN, LHTAB, EXTAB, and RHTAB. The contents of register 7 is inserted into the parameter list as the fourth parameter.

May 1983

CMD

Macro Description

Purpose: To assemble a call to the CMD subroutine. See the CMD subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] CMD reg[,len]

Parameters:

reg specifies the command to be given to CMD. It may be a command enclosed in primes, the name of a region containing the command, or the number of a register (which contains the location of the command) enclosed in parentheses.

len (optional) specifies the length of the command. It may be the name of a halfword or fullword containing the length, or the number of a register (which contains the length) enclosed in parentheses. If it is omitted, L'REG is used. It must be omitted if reg is 'text' and may not be omitted if reg specifies a register.

This macro destroys the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Examples:

```
LBL   CMD '$SET ECHO=OFF'
```

This example calls CMD with $SET ECHO=OFF as the command. This same effect can also be obtained by calling the CUINFO subroutine.

```
CMD CMDLOC
```

This example calls CMD with location CMDLOC containing a command. The length of CMDLOC defines the length of the command.

```
CMD (2),(3)
```

This example calls CMD with register 2 containing the location of a command and register 3 containing the length of the command.

CMDNOE

Macro Description

Purpose: To assemble a call to the CMDNOE subroutine. See the CMDNOE subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] CMDNOE reg[,len]

Parameters:

reg specifies the command to be given to CMDNOE. It may be a command enclosed in primes, the name of a region containing the command, or the number of a register (which contains the location of the command) enclosed in parentheses.

len (optional) specifies the length of the command. It may be the name of a halfword or fullword containing the length, or the number of a register (which contains the length) enclosed in parentheses. If it is omitted, L'REG is used. It must be omitted if reg is 'text' and may not be omitted if reg specifies a register.

This macro destroys the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Examples:

```
LBL   CMDNOE '$SET ECHO=OFF'
```

This example calls CMDNOE with $SET ECHO=OFF as the command. This same effect can also be obtained by calling the CUINFO subroutine.

```
      CMDNOE CMDLOC
```

This example calls CMDNOE with location CMDLOC containing a command. The length of CMDLOC defines the length of the command.

```
      CMDNOE (2),(3)
```

This example calls CMDNOE with register 2 containing the location of a command and register 3 containing the length of the command.

CNTRL

Macro Description

Purpose: To provide an interface between the user and the CONTROL entry in the device support routines (DSRs). This macro allows the user to execute control operations on files and devices. See the CONTROL subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype:

```
[label]  CNTRL  fdub,'text'[,ret][,EXIT=exitseq]
                [,MF=mod][,TREG=(r1,r2)]

[label]  CNTRL  fdub,info[,len][,ret][,EXIT=exitseq]
                [,MF=mod][,TREG=(r1,r2)]
```

Parameters:

fdub is either
(1) the location of a FDUB-pointer, a full-word logical I/O unit number (0 through 19), or a left-justified, 8-character logical I/O unit name,
(2) 'name', where "name" is the name of a logical I/O unit,
(3) a logical I/O unit number (0 through 19), or
(4) a register number in parentheses (the register must contain a FDUB-pointer or a logical I/O unit number).

text is the actual text of the device control information to be passed to the device support routines.

info is the location of the device control information to be passed to the device support routines. This may be either an expression or a register number in parentheses (the register must contain the location).

len (optional) is the length of the control information. This may be either an expression defining the location of a halfword containing the length, a self-defining term which is length, or a register number in parentheses (the register must contain the length). If len is omitted, L'info is assumed.

ret (optional) is the location of an area of 27 fullwords to receive the return information from the device support routines. This area will contain:

Word 1: return code from the DSR
2: length of the DSR message, or zero
3-27: DSR error message (if given)

ret may be specified as an expression or a register number in parentheses (the register must contain the location of the area). If ret is omitted, the return information will be discarded.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E is specified. If this is omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is specified in the parameter list, only the parameter list will be generated. In this case, no executable code is produced, and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or a register number in parentheses (the register must contain the location of an MF=L form). If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, GR14 and GR15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call.

Example:

```
LAB   CNTRL  8,'REW',RETINFO,EXIT=ERROR
```

In this example, a "rewind" control command is issued for logical I/O unit 8. Any return information from the device support routines will be stored in RETINFO. If a nonzero return code occurs, a branch will be made to the location ERROR.

DFAD, DFSB, DFMP

Macro Description

Purpose: To simulate extended-precision floating-point operations using the instructions available on the machine. These macros will add, subtract, or multiply two contiguous, long, floating-point registers by an extended-precision (16-byte) operand and place the result in the same two registers.

Prototype:
```
[label]  DFAD   reg,adr,temp
[label]  DFSB   reg,adr,temp
[label]  DFMP   reg,adr,temp
```

Parameters:

- reg is the first of two contiguous floating-point registers. Note that reg must be 0 or 4.
- adr is the location of the doubleword-aligned, extended-precision (16-byte) operand.
- temp is the location of a doubleword-aligned scratch area. For DFAD and DFSB, this must be 16 bytes long; for DFMP, this must be 64 bytes long.

Description: The contents of adr are added to, subtracted from, or multiplied by the contents of the long floating-point registers reg and reg+2.

An extended-precision operand may be considered as two long floating-point operands. Both operands have a characteristic and a mantissa; the characteristic of the second long operand is 14 less than the characteristic of the first long operand.

These macros inspect the value of the GBLB symbol &S370 to determine whether the program is to be run on a 360/67 or a 370. If &S370 is 0, they use the hardware operations ADD, ADDR, SDD, SDDR, MDD, and MDDR. See the description of these instructions in the section "Extensions to the System/360 Model 67 Operations" in this volume. If &S370 is 1, they use the hardware operations AXR, SXR, or MXR. These are described in IBM System/370 Principles of Operation, form number GA22-7000.

DCI, DCINIT

Macro Description

Purpose: To automatically initialize storage in a dsect.

Prototype:

```
symbol  DCI    const1,const2,...
        DCINIT dsect[,TYPE={ORG|SUBR}]
```

Parameters:

- symbol — specifies the name of the location(s) in the dsect that are to be initialized.
- const — specifies the constant values that are to be stored into the initialized locations.
- dsect — specifies the name of the dsect that contains the locations to be initialized.
- TYPE={ORG|SUBR} — specifies whether the generated initialization code is to reside in the csect at the point of the DCINIT macro call (TYPE=ORG), or is to reside in a separate subroutine which is called at the point of the DCINIT macro call (TYPE=SUBR). For TYPE=SUBR, a standard OS (I) S-type call is made; register 1 must contain the address of the dsect being initialized. A separate csect will be generated to collect the subroutine initialization code for all the DCINIT macro calls. The default is TYPE=ORG.

Description: The DCI and DCINIT macros provide a method for automatically initializing storage in a dsect. The DCI macro defines and generates the executable code to initialize the storage. The DCINIT macro specifies where the initialization is to reside.

The syntax of the DCI macro is the same as the syntax for the DC pseudo-op except that bit-length modifier (L.) is not supported.

Address-type constants (AL3 and AL4) are initialized by a series of load address (LA) and store (ST or STM) instructions. All other constants are initialized by move (MVI, MVC, or MVCL) instructions with the source being the constant value expressed as a literal. All literals will be emitted by a LTORG psuedo-op in the control section containing the initialization code.

Example: The following example illustrates the use of of the DCI and DCINIT macros.

```
                DCINIT MYDSECT
                 ...
MYDSECT   DSECT
INPAR     DCI    A(BUF,INL,MOD,LNUM)
BUF       DS     CL80
INL       DCI    Y(0,256,0)
MOD       DCI    X'80000002'
LNUM      DCI    FE3'1'
                 END
```

The above sequence generates the following code in the csect containing the DCINIT call (TYPE=ORG is defaulted).

```
                LA     R0,BUF
                LA     R1,INL
                LA     R2,MOD
                LA     R3,LNUM
                STM    R1,R3,INPAR
                MVC    INL(6),=Y(0,256,0)
                MVC    MOD(4),=X'80000002'
                MVC    LNUM(4),=FE3'1'
```

May 1983

DFIX, EFIX

Macro Description

Purpose: To convert a floating-point number (in a floating-point register) to an integer (in a general register).

Prototype:
```
[label]  DFIX   fpr,gr[,WA=wkarea]
[label]  EFIX   fpr,gr[,WA=wkarea]
```

Parameters:

- fpr is the floating-point register.
- gr is the general register.
- wkarea (optional) is a keyword parameter designating a doubleword-aligned work area of 16 bytes. If omitted, the macro will allocate an in-line work area.

The condition code is unpredictable afterwards.

Description: DFIX converts a long-precision, floating-point number (8 bytes); EFIX converts a short-precision, floating-point number (first 4 bytes of a floating-point register). The contents of the specified floating-point register are restored at the end of the macro call. Note that it is possible to convert a floating-point number that is too big to fit (as an integer) into a general register, but the results will be meaningless since in order to make the floating-point number fit into a general register, part of the number is truncated. No attempt is made to signal this as an error.

Example:
```
          LBL   DFIX  0,0
```

This example converts the 8-byte floating-point number in floating-point register 0 into an integer in general register 0.

DISMOUNT

Macro Description

Purpose: To assemble a call to the DISMOUNT subroutine.

Prototype: [label] DISMOUNT 'string'

Parameters:

string is one or more pseudodevice names (separated by blanks or commas) for the items to be released.

This macro destroys the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: A call will be assembled to the DISMOUNT subroutine. See the DISMOUNT subroutine description in MTS Volume 3, System Subroutine Descriptions. The macro generates literal constants; thus literal addressability must be preserved.

Example:

```
DMNT  DISMOUNT  '*TAPE*'
```

This example will assemble a call to DISMOUNT to release *TAPE*.

ENTER

Macro Description

Purpose: To generate prolog code for the entrance to a subroutine.

Prototype:
```
[label]  ENTER  reg[,par][,SA=savarea][,LENGTH=len]
                [,TREG=tempreg][,ID=idval][,T=type]
                [,DSECT=base13][,BASE=baseval][,LIT=NO]
```

Parameters:

- reg — is the register or list of registers to be established as a base register. None of the registers should be the same as tempreg or be registers 0 or 13. In addition, if savarea is omitted, registers 1, 14, and 15 cannot be used.
- par — (optional) is the register to copy register 1 into. This will be the pointer to the parameters. If par is specified, it cannot be the same as tempreg or reg or be register 13. If both par and savarea are specified, they cannot be registers 0, 14, or 15. If par is omitted, register 1 is not copied.
- savarea — (optional) is a keyword parameter specifying the location of a save area to use. If savarea is omitted, a call to the GETSPACE subroutine is made to get a save area of length specified by len. If savarea is *, no save area is set up.
- len — (optional) is a keyword parameter specifying the length of the save area to be obtained if savarea is omitted. If len is omitted, 72 is used.
- tempreg — (optional) is a keyword parameter specifying the temporary register to be used in the prolog code. If omitted, register 15 is used. If savarea specifies a location, tempreg should not be registers 0 or 13. If savarea is omitted, tempreg should not be registers 0, 1, or 13. tempreg is not used if savarea is *.
- idval — (optional) is the ID to be assigned to this entry. The length of the ID will be the fourth byte generated and the ID itself will start at the fifth byte. If idval is * or (*,name), then either label will be used for the ID, or if label is missing, then the name of the enclosing CSECT will be used, or

if both are missing, then name will be used. In this last case a "name CSECT" statement is also generated. If idval is $ or ($,name2), then either the name of the CSECT is used for the ID, or if the CSECT is unnamed, then name2 is used, or if name2 is missing, then label is used. In these latter two cases a CSECT statement is generated.

type (optional) is the type code to be used on the call to GETSPACE for the save area if savarea is omitted. If the T parameter is omitted, the type code of 3 is used.

base13 (optional) is the name of a DSECT describing the save area allocated if savarea is omitted. A "USING base13,13" is inserted.

baseval (optional) is a value or list of values to be loaded into the base register or registers specified by reg. If baseval is "$", the origin of the control section is used as the value to be loaded into reg.

LIT=NO (optional) specifies that no literals should be used in code generated by this macro. In this case, constants will be generated inline with a branch around them. The macro assumes that savarea is also not addressable if LIT=NO is specified.

Description: ENTER produces all of the code normally needed at the entry to a subroutine to:

(1) Save the caller's registers.
(2) Establish base registers for this program.
(3) Copy the parameter list pointer to a safe register.
(4) Establish a save area for this program.

Base registers are established as follows: Either or both of reg or baseval may be a list enclosed in parentheses. The number of base registers established is equal to the number of items in the longer list. If an entry in reg is omitted, it is assumed to be the previous register plus one. If an entry in baseval is omitted, it is assumed to be the previous value plus 4096, unless it is the first entry, in which case the location of the ENTER macro is assumed.

Examples:

```
SUB1  ENTER 12
```

This example generates an entry sequence using register 12 as the base register. A 72-byte save area is allocated by the macro.

```
SUB2  ENTER 9,SA=SAVEAREA
```

This example generates an entry sequence using register 9 as the base register and the region at location SAVEAREA as the save area.

```
SUB3  ENTER 11,TREG=12
```

This example generates an entry sequence using register 11 as the base register and register 12 as the temporary register in the sequence. A 72-byte save area is allocated by the macro.

```
SUB4     ENTER (9,10,11),SA=SAVE
```

This example generates an entry sequence using registers 9, 10, and 11 as base registers addressing SUB4, SUB4+4096, and SUB4+8192, respectively.

```
SUB5     ENTER 9,SA=SAVE,BASE=(BAS1,,BAS2)
```

This example generates an entry sequence using registers 9, 10, and 11 as base registers addressing BAS1, BAS1+4096, and BAS2, respectively.

ERROR

Macro Description

Purpose: To assemble a call to the ERROR subroutine. See the ERROR subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] ERROR

Description: This will generate

```
label  L    15,=V(ERROR)
       BALR 14,15
```

EXIT

Macro Description

Purpose: To reestablish the calling program's save area and to return with a return code in GR15 and an optional returned value in GR0.

Prototype:
```
[label]  EXIT   [rc][,rval][,MF=mod][,SPM=YES]
                [,TREG=(n,m)][,RESTORE=gr][,DROP=regs]
                [,COMPLET=NO]
```

Parameters:

- rc — (optional) is a return code to be loaded into GR15. If rc is omitted, the return code is zero. rc may be expressed as a register number in parentheses, a self-defining term, a storage location, or * meaning the return code is in register 15.
- rval — (optional) is a return value to be loaded into GR0. It may be expressed as a register number in parentheses, a self-defining term, a storage location, or * meaning the value is in register 0. If rval is omitted, the default of * is used.
- mod — (optional) If given as MF=FS, specifies that the save area pointed to by GR13 is to be released by calling FREESPAC. If given as MF=*, specifies that GR13 already points to the caller's save area.
- SPM=YES specifies that the program mask is to be restored from register 14 before returning.
- (n,m) n and m are two registers that may be used by the macro. They must not be 0, 1, 13, 14, or 15. They will only be used if MF=FS is specified. Registers 2 and 3 are used if the TREG parameter is omitted.
- gr — is the number of the first register to restore. If gr is given, registers 14 and gr through 12 are restored. If the RESTORE parameter is omitted, gr is 0 if rval is omitted and 1 otherwise.
- regs — is a list of registers to be DROPped. This list should be enclosed in parentheses and separated by commas.
- COMPLET=NO specifies that the 12th byte of the caller's save area is not to be set to X'FF'. This setting to X'FF' is normally done to indicate that this subroutine has returned.

Description: This macro will restore registers and optionally the program mask, and return to the program which called the current subroutine. If MF=* is not specified, the save areas must be properly linked on entry as is done by the ENTER macro.

Examples:

```
        EXIT  4
```

This example generates a return with a return code of 4 in register 15.

```
OUT     EXIT  0,(1)
```

This example generates a return with a return code of zero in register 15 and a return value from register 1 in register 0.

```
        EXIT  0,4
```

This example generates a return with a return code of zero in register 15 and a return value of 4 in register 0.

```
LABEL   EXIT  0,RVAL
```

This example generates a return with a return code of zero in register 15 and the return value from location RVAL in register 0.

FLOAT

Macro Description

Purpose: To convert the contents of a general register or a fullword area in storage into a floating-point number and leave the converted number in a floating-point register.

Prototype: [label] FLOAT arg1,arg2[,WA=wkarea]

Parameters:

- arg1 can either be a general register or a fullword location; if arg1 specifies a general register, the argument must be enclosed in parentheses.
- arg2 is the floating-point register into which the results are placed.
- wkarea is the user-specified doubleword scratch area. If omitted, the scratch area is generated in-line within the macro expansion.

Addressability of the literal pool is required.

After execution, the condition code will be set as follows:

0 if value=0
1 if value<0
2 if value>0

Examples:

```
FLOAT A,2
```

This example converts the contents of location A to a floating-point number and stores the result in floating-point register 2.

```
FLOAT (6),4
```

This example converts the contents of general register 6 to a floating-point number and stores the result in floating-point register 4.

FREESPAC

Macro Description

Purpose: To assemble a call to the FREESPAC subroutine. See the FREESPAC subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] FREESPAC [loc][,length][,EXIT=err]

Parameters:

- loc (optional) is the location of the space to be released or the number of a register (which contains the location) in parentheses. If this parameter is omitted or coded as *, it is assumed that the location is given in register 1.
- length (optional) is the length of the region to be released, the number (in parentheses) of a register which contains the length, or * meaning the length is in register 0. If this parameter is omitted, the length is assumed zero (release whole region).
- err is the location to branch to if the FREESPAC subroutine detects an error.

This macro destroys the contents of registers 0, 1, 14, and 15. Register 13 need not point to a save area.

Examples:

```
LAB1  FREESPAC A
```

This example calls FREESPAC with the location of the region to be released in location A.

```
LAB2 FREESPAC B,1024
```

This example calls FREESPAC with the location of the region to be released in location B. The length to be released is 1024 bytes.

```
LAB3 FREESPAC ,(2)
```

This example calls FREESPAC with the location of the region to be released in register 1. The length to be released is in register 2.

May 1983

GETSPACE

Macro Description

Purpose: To assemble a call to the GETSPACE subroutine. See the GETSPACE subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] GETSPACE [length][,T=type][,EXIT=err]

Parameters:

- length (optional) is the number of bytes of storage wanted. It may be a self-defining term, an expression, the name of a location, or a register number in parentheses. If omitted or coded as *, the length is assumed to be in register 1.
- type (optional) is a keyword parameter specifying the type of space wanted. type should be from 0 to 7. If omitted, type is assumed to be 3. See the description of the GR0 contents in the GETSPACE subroutine description in MTS Volume 3.
- err is the location to branch to if the GETSPACE subroutine detects an error.

This macro destroys the contents of registers 0, 1, 14, and 15. Register 13 need not point to a save area. The condition code may be changed.

Example:

```
LAB   GETSPACE  8192
```

This example calls GETSPACE for a region of 8192 bytes of storage.

GUSER

Macro Description

Purpose: To assemble a call to the GUSER subroutine. See the GUSER subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] GUSER reg[,regl][,modifs][,lnr][,EXIT=exitseq] [,MF=mod][,TREG=(r1,r2)]

Parameters:

- reg is the location of the region into which the record is to be read. This may be expressed as a symbol or as the number of a register (which contains the location of the region) in parentheses.
- regl (optional) specifies the place to store the length of the input upon return from the GUSER subroutine. This may be either the name of a halfword to contain the length, or the number of a register (which will contain the length) in parentheses. If omitted, the length is discarded.
- modifs (optional) stands for several parameters separated by commas. Each parameter consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). If no modifiers are given, zero is used for the modifiers parameter to GUSER. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.
- lnr (optional) specifies the line number parameter for the GUSER subroutine. It may be the location of a fullword containing the line number, a self-defining term which is the line number, or the number of a register (which contains the line number) in parentheses. If a register is specified, it will be stored in the line number parameter before the call and loaded from it after the call. If omitted, the macro generates a fullword if needed. If the @INDEXED modifier is specified in this macro or elsewhere, the value of the lnr parameter before the call to the GUSER subroutine is used. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3 for further discus-

sion. If both regl and lnr are registers, lnr should not be GR1.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E. If omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. If modifs includes NOPROMPT and/or NOTIFY, the called subroutine will destroy the contents of register 0. If MF=(E,...) is specified, the contents of registers r1 and r2 are also destroyed. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, registers 14 and 15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call. If any modifiers are given, the new modifiers completely replace the old modifiers unless the MF parameter is given as MF=(E,listadr,SEP) in which case only the modifiers specified are changed.

Examples:

```
LAB1  GUSER   REG,LEN,EXIT=EOF
```

This example calls GUSER with REG as the location of the region to be read into and LEN as the location to store the length of the record read. A nonzero return code from GUSER will cause a branch to EOF.

```
LAB2  GUSER   REGION,LENG,@I,@PFX,EXIT=DONE
```

This example calls GUSER with REGION as the location of the region to be read into and LENG as the location to store the length of the record read. The record is read with the @I and @PFX modifiers specified. A nonzero return code will cause a branch to DONE.

INSTSET

Macro Description

Purpose: To control which set of machine instructions will be available in ASMH.

Prototype:
```
[label]  INSTSET  [M67={YES|NO}][,S370={YES|NO}]
                  [,M470={YES|NO}][,M580={YES|NO}]
                  [,M4300={YES|NO}][,MODEL=model]
                  [,FEATURES=(feature[,...])]
```

Parameters:

M67={YES|NO}

M67 controls the inclusion of the System/360 Model 67 instruction set. If M67=YES is specified, the following instructions will be available: BAS, BASR, LMC, STMC, AX, DX, LX, MX, SX, ADD, MDD, SDD, ADDR, MDDR, SDDR, SLT, SWPR, and LRA. See the section "Extensions to the System/360 Model 67 Operations" in this volume and the IBM publication, IBM System/360 Model 67 Functional Characteristics (form number GA27-2719), for a description of these instructions.

M370={YES|NO}

M370 controls inclusion of the System/370 instruction set. If S370=YES is specified the following instructions will be available: AXR, CDS, CLCL, CLM, CLRIO, CS, HDV, ICM, IPK, LCTL, LRA, LRDR, LRER, MC, MVCL, MXD, MXDR, MXR, PTLB, RRB, SCK, SCKC, SIGP, SIOF, SPKA, SPT, SPX, SRP, STAP, STCK, STCKC, STCM, STCTL, STIDC, STIDP, STNSM, STOSM, STPT, STPX, and SXR. See the IBM publication, IBM System/370 Principles of Operation (form number GA22-7000), for a description of these instructions.

M470={YES|NO}

M470 controls the inclusion of instructions peculiar to the Amdahl 470 series of machines, all of which are privileged. If M470=YES is specified the following instructions will be available: LFCR, STFCR, PPG, and PSU.

M580={YES|NO}

M580 controls the inclusion of instructions peculiar to the Amdahl 580 series of machines. There are currently no instructions of this type, therefore this parameter has no effect.

M4300={YES|NO}

M4300 controls the inclusion of instructions peculiar to the IBM 4300 series of machines. If M4300=YES is specified, the following instructions will be available: CLRP, CTP, DEP, DCTP, IPB, LFI, MAD, MADS, MUN, MVCIN, RSP, SPB, and STCAP. See the IBM publication, IBM 4300 Processors Principles of Operation for ECPS:VSE Mode (form number GA22-7070), for a description of these instructions.

MODEL=model

MODEL controls the inclusion of instructions based on the model number of the machine. This allows a finer degree of control than the more general, machine-series specification indicated by M67, S370, M470, M580, or M4300. The instruction set included for a particular mode number is that of the base machine; i.e., without any features which are considered optional on that model. Valid model numbers are:

Model	Machine Specified
67	IBM System/360 Model 67
115	IBM System/370 Model 115
125	IBM System/370 Model 125
135	IBM System/370 Model 135
135-3	IBM System/370 Model 135-3
138	IBM System/370 Model 138
145	IBM System/370 Model 145
145-3	IBM System/370 Model 145-3
148	IBM System/370 Model 148
155	IBM System/370 Model 155
158	IBM System/370 Model 158
158-3	IBM System/370 Model 158-3
165	IBM System/370 Model 165
168	IBM System/370 Model 168
168-3	IBM System/370 Model 168-3
195	IBM System/370 Model 195
3031	IBM 3031 (all submodels)
3032	IBM 3032 (all submodels)
3033	IBM 3033 (all submodels)
3081	IBM 3081 (all submodels)

3083	IBM 3083 (all submodels)
3084	IBM 3084 (all submodels)
3090	IBM 3090 (all submodels)
4321	IBM 4321 (all submodels)
4331	IBM 4331 (all submodels)
4341	IBM 4341 (all submodels)
4361	IBM 4361 (all submodels)
4381	IBM 4381 (all submodels)
5840	Amdahl 5840
5850	Amdahl 5850
5860	Amdahl 5860
5867	Amdahl 5867
5868	Amdahl 5868
5870	Amdahl 5870
5880	Amdahl 5880
5890	Amdahl 5890
470/V6	Amdahl 470/V6
470/V7	Amdahl 470/V7
470/V8	Amdahl 470/V8

FEATURES=(feature[,...])

FEATURES controls the inclusion of instructions on the basis of which facility they are a part of. FEATURES modifies the instruction set defined by the MODEL parameter. It may only be specified when the MODEL parameter is also specified.

Any combination of features may be included or excluded, regardless of whether that feature is available on the machine implied by MODEL. A feature is included by specifying its name, chosen from the following list, for "feature". A feature is excluded by specifying its name preceded by "-", "¬", or "NO", for "feature".

Feature	Machine Facility Specified
AP	Multiprocessing
BAS	Branch and save
BRANCH_AND_SAVE	Branch and save
CHANNEL_SET_SWITCHING	Channel set switching
CLOCK_COMPARATOR	CPU timer and clock comparator
CONDITIONAL_SWAPPING	Conditional swapping
COMPARE_AND_SWAP	Conditional swapping
CPU_TIMER	CPU timer and clock comparator
CS	Conditional swapping
CSS	Channel set switching
DAS	Dual address space
DAT	Translation

DC	Direct control
DIRECT_CONTROL	Direct control
DUAL_ADDRESS_SPACE	Dual address space
ECPS	Extended control program support
EXPANDED_STORAGE	Instructions to access expanded storage
EXT	Extended
EXTENDED_FACILITY	Extended
EXTENDED_ARCHITECTURE	System/370 Extended Architecture
EXTENDED_PRECISION	Extended-precision floating point
FXP	Extended-precision floating point
ISKE	Storage-key-instruction extensions
MAD	Multiply and add
MOVE_INVERSE	Move inverse
VECTOR_FACILITY	IBM 3090 vector instructions

IOH Macros

Macro Description

Purpose: To generate calls to IOH to perform formatted input and output.

Prototype:

```
[label]  RDFMT    fmt[,(par,...)]
[label]  PRFMT    fmt[,(par,...)]
[label]  PCFMT    fmt[,(par,...)]
[label]  WRFMT    fmt[,(par,...)]
[label]  SERFMT   fmt[,(par,...)]
[label]  GUSFMT   fmt[,(par,...)]
```

Parameters:

- fmt specifies the location of the IOH format. This must be given as a symbolic expression. See the section "IOH" in this volume for a description of the format language.
- par specifies one simple or block parameter giving the location to be read or written. If a simple parameter is desired, this must be specified as a symbolic expression. If a block parameter is desired, this must be specified as two symbolic expressions separated by ",...,"; for example,

```
A,...,A+20
```

These macros destroy the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: The above macros are used to call IOH from assembly language programs. This description covers only the most elementary usage omitting many additional parameters which may be specified, and several other related macros. For a complete description of IOH, see the section "IOH" in this volume.

When one of these macros is executed, IOH will be called to perform input or output according to the format given by fmt into or from the locations specified by par. Any number of simple or block parameters may be specified, and input or output will continue until a parameter specified as "0" is encountered. For this reason, the last par should be given as "0" to terminate input or output.

By using some of the more advanced features of these macros, it is possible to compute dynamically the parameters to be used, specify parameters relative to base registers, etc. Those users who need the advanced features should see the section "IOH."

Examples:

```
RDLBL  RDFMT   INFMT,(CNT,A,...,A+10*4,0)
         .
         .
INFMT  DC      C'I,11WF*'
CNT    DS      F
A      DS      11E
```

This example will read 1 fullword integer and 11 fullword floating-point numbers in free format.

```
PRLBL  PRFMT   OFMT,(NUM,RESULT,C,...,C+5*4,0)
         .
         .
NUM    DS      F
RESULT DS      E
C      DS      6E
OFMT   DC      C'"-CASE ",I5,"RESULTS ",7WF6.2*'
```

This example will print 1 fullword integer and 7 fullword floating-point numbers plus the 2 comments in the format specification.

KWLHT

Macro Description

Purpose: To build a left-hand side entry for subroutine KWSCAN. See the KWSCAN subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype:
```
[label]  KWLHT    rht,exec[,keyword[,NOSPELLCOR]]
                  [,MINLEN=len]
```

Parameters:

- rht — specifies the right-hand side index. If RHTABLE=RHT has been specified on a previous KWSET macro, "rht-RHT" is assembled as the right-hand side index.
- exec — specifies the execute-table index. If EXTABLE=EXT has been specified on a previous KWSET macro, "exec-EXT" is assembled as the execute-table index.
- keyword — specifies the left-hand side keyword. This may be enclosed in primes; e.g., either KEY or 'KEY' may be used. If omitted, the macro generates a null left-hand side, implying the degenerate form "RHSide".
- NOSPELLCOR — specifies that spelling correction is to be suppressed for the left-hand side entry (by inclusion of the X'FE' control code).
- len — specifies an explicit minimum initial sub-string length for the left-hand side entry (by inclusion of the X'FD' control code).

Examples:
```
LBL     KWLHT  SPACERHT,0,SPACE
```

This example generates a left-hand table entry with SPACERHT as the right-hand side index and 0 as the execute-table index. The keyword is SPACE.

```
        KWLHT  JUNK,8
```

This example generates a null left-hand side with JUNK as the right-hand side index and 8 as the execute-table index.

KWRHT

Macro Description

Purpose: To build a right-hand side entry for KWSCAN. See the KWSCAN subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] KWRHT type[,exec[,...]]

Parameters:

type specifies the right-hand side type. See below for the description of the types available. Several of the type parameters have alternate names.

exec specifies the execute-table index. The symbol EXTABLE may be set by the KWSET macro as the execute table, in which case "exec-EXTABLE" instead of "exec" is assembled. exec may be specified as (exec,Nexec) in various types such as NEGLIT, NEGSTR, and NEGCHARS. This allows two execute-table indices. For several types, exec is omitted.

Types:

CHARACTERS or
CHARS KWRHT CHARS,exec[[,min],max]

The right-hand side is taken as an arbitrary character string, possibly subject to minimum (min) and maximum (max) length restrictions.

```
KWRHT CHARS,4,1,17
```

The above example matches the right-hand side, provided that it is 1 to 17 characters long.

DELIMITERS or
DCHARS KWRHT DCHARS,exec,min,max,dlmtrs

Delimited character strings are initiated and terminated by one of the delimiters dlmtrs. Double instances of delimiters are interpreted as single instances of the delimiters. The delimited character strings are limited by min, the minimum permissible length, and max, the maximum permissible length (less than 128).

```
KWRHT DCHARS,STRING,1,127,''''
```

The above example takes character strings that are of length 1 to 127 and are delimited by primes.

END KWRHT END

This terminates the search of the right-hand table and forces the scan for a keyword match to fail.

FDNAME KWRHT FDNAME,exec[,N]

The right-hand side is interpreted as an MTS FDname, or concatenation of FDnames, and a FDUB is acquired for it. If ",N" is specified, no FDnames specifying explicit concatenation are matched.

FLAGGEDHEX or
FHEX KWRHT FHEX,exec

The right-hand side is interpreted as a flagged hexadecimal number of 8 digits maximum, expressed in the form X'number'.

FILTER KWRHT FILTER,(POS1,...,POSn)

This specifies the separator filter between the left-hand and right-hand sides. POS1,...,POSn specifies the ordinal positions of the separators in the list passed as the slist parameter or implied by sws bits 20 and 21 having value 01 (see the KWSCAN description in MTS Volume 3).

```
KWRHT FILTER,(0,6)
```

The above example lets through only "=" and degenerate keywords, provided that bits 20 and 21 are set to 01 in sws. Position 0 specifies degenerate keywords and position 6 refers to "=".

FLOAT KWRHT FLOAT,exec[,(CODE1,NR1),...,(CODEn,NRn)]

This specifies a long, real, floating-point number. The right-hand side value is interpreted as a FORTRAN-style long, real number, optionally followed by a scale factor. Operations (CODEi,NRi) may be performed on the number. CODEi must be a single character, and NRi a valid number. Operations are performed according to the following operation codes:

Code	Action
>	The right-hand side value is compared to the operand value. If the right-hand side

value is less, the right-hand side match fails.

<	The right-hand side value is compared to the operand value. If the right-hand side value is greater, the right-hand side match fails.
*	The right-hand side value is multiplied by the operand value.
/	The right-hand side value is divided by the operand value.
Others	The operation code character is interpreted as an optional scale factor, which, if present at the end of the right-hand value, causes the value to be multiplied by the operand value.

```
KWRHT FLOAT,EPSILON
```

The above example takes the right-hand side as a floating-point number, which is passed to the subroutine EPSILON.

```
KWRHT FLOAT,MINUTES,(>,0),(S,.0166666667),(<,1000)
```

The above example takes a floating-point number which must be positive, divides it by 60 if expressed in terms of seconds, and then checks if it is less than 1000.

HEXADECIMAL or
HEX `RHT HEX,exec`

The right-hand side is interpreted as a hexadecimal number up to 8 hexadecimal digits long.

IGNORE `KWRHT IGNORE`

The entire keyword expression is ignored. No instructions in the execute table are performed.

INSET `KWRHT INSET,exec,min,max,literal`

All the characters constituting the keyword expression right-hand side must be members of a given set of characters as specified in literal. min and max must be given and refer to the minimum and maximum permissible lengths of the right-hand side.

```
KWRHT INSET,EBMETM,0,7,'WHTR$'
```

In the above example, only characters W, H, T, R, and $ are allowed in the right-hand side.

INTEGER KWRHT INTEGER,exec[,(CODE1,NR1),...,(CODEn,NRn)]

The right-hand side is interpreted as an optionally signed integer number with up to 10 decimal digits and possibly followed by a scale factor. NRi must be in the range of (-2147483648, 2147483647), inclusive. (CODEi, NRi) are the same as in type=FLOAT, except that NRi is taken as integer and that NRi can be expressed as an 360/370 Assembler expression.

```
KWRHT INTEGER,0,(M,60),(<,23*60)
```

The above example takes a number as expressed in the right-hand side. If the right-hand side is followed by the scale factor M, the number is multiplied by 60. The number is then compared with 23*60.

LINENUMBER or
LINENR KWRHT LINENR,exec[,(CODE1,NR1),...,(CODEn,NRn)]

The right-hand side is interpreted as MTS line number, an optionally signed number with 7 integral digits and 3 fractional digits. This is then multiplied by 1000 to remove any fractional digits. (CODEi,NRi) are as in type INTEGER.

```
KWRHT LINENR,0,(S,1),(M,60),(<,27962026),(>,1),
    (*,768),(/,10)
```

The above example takes the right-hand side as expressed in terms of seconds "S", multiplies by 60 if it is followed by "M", compares it with the values 27962026 and 1, then multiplies it by 768 and divides the product by 10. Finally, the result is multiplied by 1000.

LITERAL or
LIT KWRHT LIT,exec,string

The right-hand side is matched against a specified character string <u>string</u>. If there is a match, the instructions are then executed. The string may optionally be delimited by primes.

```
KWRHT LIT,0,ON
KWRHT LIT,4,'OFF'
```

In the above example, the right-hand side is compared with strings ON and OFF. If ON, the instructions at offset 0 are executed; otherwise, if OFF, the instructions at offset 4 are performed.

LITSUBSTR or
LITSUB KWRHT LITSUB,exec,min,string

The right-hand side must be a string of characters not less than "min", which must be an initial substring of the given "string". If "min" is zero, no restriction on the string length is imposed.

```
KWRHT LITSUB,STOPPER,2,'STOP'
```

In the above example, if the right-hand side is "ST", "STO", or "STOP", the instructions at STOPPER are performed. If the right-hand side is "S" or "STOPPER", the instructions are not performed.

NEGCHARACTERS or
NEGCHARS KWRHT NEGCHARS,(exec,Nexec)[[,min],max]

This is the same as in type CHARS, except they may be prefixed by one of "-", "¬", "NO", or "N". If not prefixed, exec is performed. Otherwise, Nexec is performed. Optionally, the minimum and maximum permissible lengths may be specified.

```
KWRHT NEGCHARS,(YES,NO),1,3
```

The above example executes YES if the right-hand side is not preceded by a negative prefix; otherwise, it executes NO. Only characters up to 3 in length may be used.

NEGLITERAL or
NEGLIT KWRHT NEGLIT,(exec,Nexec),string

As in type LIT, except that string may be preceded by a negative prefix.

```
KWRHT NEGLIT,(HEAD,NOHEAD),'HEAD'
```

In the above example, if the right-hand side is HEAD, the instructions at HEAD are performed. If the right-hand side is NOHEAD, the instructions at NOHEAD are performed.

NEGLITSUBSTR or
NEGLITSUB KWRHT NEGLITSUB,(exec,Nexec),min,string

As in type LITSUB, except that string may be preceded by a negative prefix.

NEGSUBSTR or
NEGSTR KWRHT NEGSTR,(exec,Nexec),string

As in type SUBSTR, except that string may be preceded by a negative prefix.

NORHS or
NORHT KWRHT NORHT,exec

If there is no right-hand side, the instructions at exec are performed.

NOSPELLCOR

This specifies that spelling correction is to be suppressed for the next right-hand side entry (by inclusion of the X'FB' control code).

NOTINSET KWRHT NOTINSET,exec,min,max,literal

As in type INSET, except that the characters constituting the keyword expression right-hand side may not contain any of the characters in a given set.

PAR KWRHT PAR,exec

The right-hand side is taken as the remainder of the input string.

PARENTHESES or
PARENS KWRHT PARENS

This processes parenthesized right-hand sides and causes the current keyword expression right-hand side to be treated as a parenthesized list of right-hand sides if such a list appears. For example, INFO=(SIZE,TYPE) would be processed as if INFO=SIZE,INFO=TYPE had been given.

POP KWRHT POP

This aborts the right-hand table search and forces the keyword scanner to reject the match of the keyword left-hand side, and to continue scanning for an alternate match to the left-hand side following the point in the left-hand table at which the previous left-hand side match was found.

SUBSTR KWRHT SUBSTR,exec,string

The right-hand side must begin with the string of characters specified in string.

The following example draws from the MTS $FILESTATUS command. It processes:

```
NAME=filename, filename
HEADING=ON, HEADING=OFF, HEAD, NOHEAD
OUTFORM=COL..., OUTFORM=KEY..., OUTFORM=LABEL...,
OUTFORM=PACK..., COL..., KEY..., LABEL..., PACK...
SIZE>=x, SIZE<=x, SIZE=x, SIZE<x, SIZE>x,
SIZE>=xP, SIZE<=xP, SIZE=xP, SIZE<xP, SIZE>xP

(This is a small subset of the parameters of the
$FILESTATUS command).
```

```
          KWSET   RHTABLE=RHT
          MVI     NAMEF,0             Initialize flag
TRYAGAIN  CALL    KWSCAN,(LHTL,LHT,EXT,STR,RHT,STRL,SWS,RVEC)
          LTR     15,15
          BZ      OK                  -> All OK.
          CLC     =F'1',RVEC
          BE      ABORT               -> User said to CANCEL it.
          CLC     =F'3',RVEC
          BNE     VERYBAD             -> Unexpected return code
          SERCOM  ' TRY AGAIN.'
          B       TRYAGAIN            -> Sic

LHTL      DC      Y(RHT-LHT)          Length of left-hand table
          SPACE   3
LHT       KWLHT   JUNK,0,'OUTFORM'
          KWLHT   HEAD,HEADE-EXT,'HEADING'
          KWLHT   NAME,NAMEE-EXT,'NAME'
          KWLHT   SIZE,SIZEE-EXT,'SIZE'
          KWLHT   JUNK,0              Null left-hand side
RHT       EQU     *
HEAD      KWRHT   FILTER,(6)          Only let through "="
          KWRHT   LIT,0,'ON'          HEADING=ON
          KWRHT   LIT,4,'OFF'         HEADING=OFF
          KWRHT   END
SIZE      KWRHT   FILTER,(1,2,4,5,6) Don't let null left-hand
*                                     sides or SIZE¬=xx through
*                                     here
          KWRHT   LINENR,0,(P,1)      SIZE (>=,<=,>,<,=)xxxP
          KWRHT   END
NAME      KWRHT   FILTER,(6)          Only let through "="
          KWRHT   CHARS,0,1,17        NAME=<1 TO 17 characters>
          KWRHT   END
JUNK      KWRHT   FILTER,(0,6)        Only let through "=" and
*                                     degenerates
          KWRHT   SUBSTR,OUTFE-EXT,'COL' OUTFORM=COL or COL
          KWRHT   SUBSTR,OUTFE-EXT+4,'KEY' OUTFORM=KEY or KEY
          KWRHT   SUBSTR,OUTFE-EXT+8,'LABEL' OUTFORM=LABEL or
*                                     LABEL
```

```
          KWRHT   SUBSTR,OUTFE-EXT+12,'PACK' OUTFORM=PACK or PACK
          KWRHT   FILTER,(0)        Only let null left-hand
*                                  side through
          KWRHT   NEGLIT,(HEADE-EXT,HEADE-EXT+4),'HEAD'
*                                  HEAD or NOHEAD
          KWRHT   CHARS,NAMEE-EXT,1,17 <filename>
          KWRHT   END
          SPACE   5
EXT       DS      0H
HEADE     MVI     HEADF,1           Header
          MVI     HEADF,0           No header
NAMEE     BAL     15,*+4            Make this a subroutine
          TM      NAMEF,1           Already have a name?
          BO      16(,15)           -> Yup, user blew it
          OI      NAMEF,1           Remember name was saved
          EX      1,FILEMVC         Save name
          BR      15                -> To KWSCAN
FILEMVC   MVC     FILENAME(0),0(2)
SIZEE     BAL     15,*+4            Make this a subroutine
          STC     5,RELATION        Save relational character
          ST      2,SIZEVAL         Save size value
          BR      15                -> To KWSCAN
OUTFE     MVI     FORMF,0           Select heading format
          MVI     FORMF,1
          MVI     FORMF,2
          MVI     FORMF,3
          SPACE   5
HEADF     DS      X
NAMEF     DS      X
FILENAME  DS      CL17
RELATION  DS      X
SIZEVAL   DS      F
FORMF     DS      X
STR       DC      CL80'OUTFORM=COL,JUNK,SIZE>5P,NOHEAD'
STRL      DC      H'80'
SWS       DC      X'0000E427'       Correct spelling, RVEC
*                                   format, relational
*                                   separators, uppercase,
*                                   print, prompt, multiple
*                                   keywords
RVEC      DS      27F
```

KWSET

Macro Description

Purpose: To set the symbols for the macros KWLHT and KWRHT.

Prototype:

```
KWSET RHTABLE=rht,EXTABLE=ext,LHTL=len
```

Parameters:

- rht — specifies the right-hand side table. See the KWLHT macro.
- ext — specifies the execute table. See both the KWLHT and KWRHT macros.
- len — specifies that in the KWLHT macro the displacements of right-hand side and execute-table indices are to be len bytes long. The legal values for len are 1 or 2; the default is 1.

Examples:

```
KWSET RHTABLE=RHT,EXTABLE=EXEC
```

This sets RHTABLE to the right-hand side table RHT, and EXTABLE to the execute table EXEC.

```
KWSET LHTL=2
```

This example resets RHTABLE and EXTABLE to null, and specifies that the displacements in KWLHT macro are to be two bytes long.

May 1983

LABEL

Macro Description

Purpose: To define a label as a machine instruction.

Prototype: [label] LABEL

Description: If the label is specified, the macro expansion will generate:

```
            DS     0H
label       EQU    *,,C'I'
```

LITADDR

Macro Description

Purpose: To set a switch to indicate to the following macros if literals are addressable.

Prototype: LITADDR [YES|NO]

Description: This macro sets the Global SETB (GBLB) switch &LITADDR to 1 if the parameter to the macro is YES or if there is no parameter. This switch is tested by certain macros, such as CALL, to see whether it can generate a literal instead of an in-line adcon.

May 1983

MAXxx, MINxx

Macro Description

Purpose: To pick the maximum (or minimum) from the list of arguments.

Prototype: Short form:

```
[label]   MAXxx   x,y
[label]   MINxx   x,y
```

Long form:

```
[label]   MAXxx   result,a,b,...,z
[label]   MINxx   result,a,b,...,z
```

Parameters:

MAXxx or MINxx	specifies the macro name.
x	specifies the first argument and the result.
y	specifies the second argument.
result	specifies the result.
a,b,...,z	specifies the list of arguments.

The condition code is unpredictable afterwards.

Description: The macros MAXxx (or MINxx) place the maximum (or minimum) of all arguments in the result (either "x" or "result" according to which form of macros is being used). The short form is used when there are only two macro operands; in this case, the first operand is compared with the second operand, and the maximum (or minimum) of the two operands is placed in the first operand. The long form is used when there are more than two macro operands. Here, the first operand is considered as the "result" while all operands are arguments. All arguments must be of the same type for any MAXxx or MINxx macro. If an argument is enclosed within parentheses, then the value is in the specified general register (MAX/MIN, MAXH/MINH, MAXL/MINL) or in the specified floating-point register (MAXE/MINE, MAXD/MIND). The result, which is the maximum (or minimum) of all arguments, is placed in "result" or "x", which must be a general register for MAX/MIN, MAXH/MINH, MAXL/MINL, or a floating-point register for MAXE/MINE, MAXD/MIND, or a packed-decimal datum for MAXP/MINP, or a character field for MAXC/MINC. All arguments in MAXP/MINP macros must fit the "result" or a data overflow will result. All

arguments in MAXC/MINC macros must be character fields of same length as the "result", and this length must be less than or equal to 256.

Note that for the short form, only two operands are compared and the result is placed in the first operand. It is necessary that the first operand must conform to the type of the result.

Macro	Type of operands	Type of result
MAX MIN	Fullword integer or general register	General register
MAXH MINH	Halfword integer or general register	General register
MAXL MINL	Logical fullword or general register	General register
MAXE MINE	Short floating register or fullword	Short floating register
MAXD MIND	Long floating register or doubleword	Long floating register
MAXP MINP	Packed-decimal data	Packed-decimal datum
MAXC MINC	Character fields	Character field

Examples:

```
LM    0,1,=CL8'SPRINT'  Get information for SPRINT
CALL  GDINFO
USING GDDSECT,1         Now using GDINFO dsect
LH    2,GDOUTLEN        Obtain maximum output length
MAXH  2,=H'20'          If less than 20, set to 20
MINH  2,=H'121'         If more than 121, set to 121
DROP  1
STH   2,OUTLEN          Set output length
...
COPY  *GDINFODSECT
```

This example obtains the maximum output length for the logical I/O unit SPRINT and sets it within the bounds of (20,121).

```
MAXC  0(8,8),A,B,C,D    Set maximum of A, B, C, D
MINC  8(8,8),A,B,C,D    Set minimum of A, B, C, D
```

This example finds the maximum and minimum of four character fields of length 8. Both the maximum and minimum are placed in two doublewords pointed to by register 8.

```
     MIN   15,F1,F2,(9)
```

This example obtains the minimum of fullword integers F1, F2, and register 9. The minimum is put in general register 15.

```
     LM    3,5,=A(ARR,4,ARR+(10-1)*4)
     LE    0,0(0,3)          Load first array element
     DO    BXLE=(3,4)        Loop for all other elements
     MAXE  0,0(0,3)          Get maximum of two values
     ENDDO ,
     STE   0,MAX             Store the maximum.
     ...
MAX  DS    E                 Holds maximum in the array
ARR  DS    10E               Ten floating-point elements
```

This example finds the maximum of short floating-point array and stores it in the variable MAX.

MOUNT

Macro Description

Purpose: To assemble a call to the MOUNT subroutine.

Prototype: [label] MOUNT 'string'

Parameters:

string is the character string for one or more mount requests separated by semicolons (see the $MOUNT command description in Volume 1, The Michigan Terminal System).

This macro destroys the contents of registers 0, 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: A call to the MOUNT subroutine will be assembled. A description of the MOUNT subroutine is given in MTS Volume 3, System Subroutine Descriptions. The macro generates literal constants, thus literal addressability must be preserved.

Example:

```
STA  MOUNT 'C9999 ON 9TP *T* RING=IN ''TAPE 462'''
```

This example mounts the magnetic tape C9999 on a 9-track tape drive. The tape is mounted with the file-protect ring in and is assigned the pseudodevice name *T*. The tape ID is TAPE 462. Note that the double primes are required in the macro call to produce single primes in the character string used to call MOUNT.

May 1983

MSG, PMSG, PHRASE

Macro Description

Purpose: To assemble and print messages. MSG assembles a message. PMSG assembles and prints a message. PHRASE assembles partial (unterminated) messages.

Description: The description of these macros is given in the section "The Message Macros" in this volume.

May 1983

MTS

Macro Description

Purpose: To assemble a call to the MTS subroutine. See the MTS subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] MTS

This macro destroys the contents of registers 14 and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: This will generate

```
label  L    15,=V(MTS)
       BALR 14,15
```

May 1983

MTSCMD

Macro Description

Purpose: To assemble a call to the MTSCMD subroutine. See the MTSCMD subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] MTSCMD reg[,len]

Parameters:

reg specifies the command to be given to MTSCMD. It may be a command enclosed in primes, the name of a region containing the command, or the number of a register (which contains the location of the command) enclosed in parentheses.

len (optional) specifies the length of the command. It may be the name of a halfword or fullword containing the length, or the number of a register (which contains the length) enclosed in parentheses. If it is omitted, L'REG is used. It must be omitted if reg is 'text' and may not be omitted if reg specifies a register.

This macro destroys the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Examples:

```
LBL   MTSCMD '$RESTART SPRINT=-X'
```

This example calls MTSCMD with $RESTART SPRINT=-X as the command.

```
MTSCMD CMD
```

This example calls MTSCMD with location CMD containing a command. The length of CMD defines the length of the command.

```
MTSCMD (2),(3)
```

This example calls MTSCMD with register 2 containing the location of a command and register 3 containing the length of the command.

May 1983

MTSMODS

Macro Description

Purpose: To assemble a word of modifier bits for the I/O routines.

Prototype: [label] MTSMODS mods

Parameter:

mods is either a single item, or else a list of items, enclosed in parentheses and separated by commas. Each item consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). For a list of names and their meanings, see the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.

Examples:

```
MODI    MTSMODS @I

OUTMOD  MTSMODS (@I,@PFX,@¬TRIM)
```

May 1983

QUIT

Macro Description

Purpose: To assemble a call to the QUIT subroutine. See the QUIT subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] QUIT [WHO=tb,][WHEN=nl]

Parameters:

- tb (optional) either BATCH (the default) indicating to call QUIT in batch mode only, or ALL indicating to call QUIT always.
- nl (optional) is either NOW (the default) indicating that the SYSTEM subroutine is to be called after a return from QUIT, or LATER indicating an omission of the call to SYSTEM.

This macro destroys the contents of registers 14 and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Example:

```
          LBL   QUIT  WHEN=LATER
```

This example calls QUIT; the call to SYSTEM is omitted. This call is effective for batch mode only.

```
          LBL1  QUIT  WHO=ALL
```

This example calls QUIT and then calls SYSTEM immediately afterwards. This is effective for both terminal and batch mode.

READ

Macro Description

Purpose: To assemble a call to the READ subroutine. See the READ subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] READ unit,reg[,regl][,modifs][,lnr]
[,EXIT=exitseq][,MF=mod][,TREG=(r1,r2)]

Parameters:

- unit — specifies the corresponding parameter to be given to the READ subroutine. This parameter is either a number from 0 through 19, the name of a logical I/O unit enclosed in primes, the location of a fullword-aligned fullword containing a FDUB-pointer, or the number of a register (which contains a FDUB-pointer) in parentheses.
- reg — is the location of the region into which the record is to be read. This may be expressed as a symbol or as the number of a register (which contains the location of the region) in parentheses.
- regl — (optional) specifies the place to store the length of the input upon return from the READ subroutine. This may be either the name of a halfword to contain the length, or the number of a register (which will contain the length) in parentheses. If omitted, the length is discarded.
- modifs — (optional) stands for several parameters separated by commas. Each parameter consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). If no modifiers are given, zero is used for the modifiers parameter to READ. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.
- lnr — (optional) specifies the line number parameter for the READ subroutine. This may be the location of a fullword containing the line number, a self-defining term which is the line number, or the number of a register (which contains the line number) in parentheses. If a register is specified, it will be stored in the line number parameter

before the call and loaded from it after the call. If omitted, the macro generates a fullword if needed. If the @INDEXED modifier is specified in this macro or elsewhere, the value of the lnr parameter before the call to the READ subroutine is used. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3 for further discussion of this. If both regl and lnr are registers, lnr should not be GR1.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E. If it is omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. If modifs includes NOPROMPT and/or NOTIFY, the called subroutine will destroy the contents of register 0. If MF=(E,...) is specified, the contents of registers r1 and r2 are also destroyed. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced, and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, registers

14 and 15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call. If any modifiers are given, the new modifiers completely replace the old modifiers unless the MF parameter is given as MF=(E,listadr,SEP) in which case only the modifiers specified are changed.

Examples:

```
      LAB1  READ   5,INREG,L,EXIT=(EOF,OUCH)
```

This example calls READ, specifying that a record is to be read from logical I/O unit 5 into the region at the location INREG and that the length of the record is to be stored in location L. A branch is made to EOF upon a return code of 4 from READ; a branch is made to OUCH upon a return code of 8 or greater.

```
      LAB2  READ   'SCARDS',REGION,LENG,@I,@PFX,EXIT=DONE
```

This example calls READ, specifying that a record is to be read from SCARDS into the region at the location REGION and that the length of the record is to be stored in the location LENG. The record is read with the @I and @PFX modifiers specified. A branch is made to DONE upon a nonzero return code from READ.

May 1983

REQU

Macro Description

Purpose: To generate EQU statements for the general registers or floating-point registers.

Prototypes:
```
REQU  [TYPE={HEX|DEC|BOTH}][,PFX=prefix]
REQU  TYPE={FRS|FPR}[,PFX=prefix]
```

Description: The first form of REQU generates 16 EQU statements equating symbolic register names to the values 0 through 15. The PFX keyword specifies the symbolic prefix to be used. The default for prefix is R. Thus if the default PFX is used, the macro generates statements equating R0 through RF or R0 through R15 to the values 0 through 15, respectively.

The code generated (assuming PFX=R) is

```
R0    EQU   0
R1    EQU   1
R2    EQU   2
R3    EQU   3
R4    EQU   4
R5    EQU   5
R6    EQU   6
R7    EQU   7
R8    EQU   8
R9    EQU   9
```

plus the following if val is HEX, BOTH, or omitted

```
RA    EQU   10
RB    EQU   11
RC    EQU   12
RD    EQU   13
RE    EQU   14
RF    EQU   15
```

plus the following if val is DEC or BOTH

```
R10   EQU   10
R11   EQU   11
R12   EQU   12
R13   EQU   13
R14   EQU   14
R15   EQU   15
```

The second form of REQU generates 4 EQU statements equating symbolic register names to the values 0, 2, 4, and 6. The PFX keyword specifies the symbolic prefix to be used. The default for prefix is FR.

The code generated (assuming PFX=FR) is

```
FR0   EQU   0
FR2   EQU   2
FR4   EQU   4
FR6   EQU   6
```

RETURN

Macro Description

Purpose: To return control to the calling program and to signal normal termination of the returning program.

Prototype: [label] RETURN [(r1[,r2])][,T][,RC=code]

Parameters:

- r1,r2 (optional) is the range of registers to be restored from the save area pointed to by the address in GR13. The registers should be specified to cause the loading of registers 14, 15, and 0 through 12 when used in a LM instruction. If r2 is not specified, only the register specified by the r1 operand is loaded. If the operand is omitted, the contents of the registers are not altered.
- T (optional) causes the control program to flag the save area used by the returning program. A byte containing all 1's is placed in the high-order byte of word 4 of the save area after the registers have been loaded.
- code (optional) is the return code to be passed to the calling program. The return code should have a maximum value of 4095; it will be placed right-adjusted in GR15 before the return is made. If RC=(15) is coded, it indicates that the return code has been previously loaded into GR15; in this case the contents of GR15 are not altered or restored from the save area. (If this operand is omitted, the contents of GR15 are determined by the r1,r2 operands.)

Description: The return of control by the RETURN macro instruction is always made by executing a branch instruction using the address in GR14. This macro can be written to restore a specified range of registers, provide the proper return code in GR15, and flag the save area by the returning program. See the "Calling Conventions" description in MTS Volume 3, System Subroutine Descriptions, for a further explanation of save areas and their formats.

Examples:

```
LAB1  RETURN (14,12),RC=4
```

This example generates a return sequence which restores registers 14 through 12 from the save area specified by register 13. A return code of 4 is given in register 15.

```
LAB2  RETURN (5,10),T
```

The example generates a return sequence which restores registers 5 through 10 from the save area pointed to by register 13. The save area is flagged in word 4.

```
LAB3  RETURN (14,12),T,RC=(15)
```

This examples generates a return sequence which restores registers 14 through 12 from the save area pointed to by register 13. The save area is flagged in word 4. Register 15 is not altered.

May 1983

REWIND

Macro Description

Purpose: To rewind a magnetic tape or file.

Prototype: [label] REWIND ldn

Parameters:

ldn is a logical I/O unit number from 0 to 19, 'SCARDS', 'SPRINT', 'SPUNCH', 'SERCOM', or 'GUSER' corresponding to the appropriate logical I/O unit specification on a $RUN command, or an FDUB-pointer.

This macro destroys the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: REWIND assembles a call to the subroutine REWIND#. See the REWIND# subroutine description in MTS Volume 3, System Subroutine Descriptions.

Examples:

```
LAB1  REWIND 0
```

This example calls REWIND#, specifying logical I/O unit 0 as the logical I/O unit to be rewound.

```
LAB2  REWIND 'SPRINT'
```

This example calls REWIND#, specifying SPRINT as the logical I/O unit to be rewound.

```
LAB3  REWIND FDUB
```

This example calls REWIND#, specifying that the tape or file referred to by the FDUB-pointer in location FDUB is to be rewound.

SAVE

Macro Description

Purpose: To save the contents of a specified set of general registers in a save area provided by the user.

Prototype: [label] SAVE (r1[,r2]),[T][,name]

Parameters:

r1,r2 is the range of general registers to be stored in the save area at the address contained in GR13. The registers should be specified so that they are stored in the order 14, 15, and 0 through 12 when used in a STM instruction. The registers are stored in words 4 through 18 of the save area. If only one register is specified, only that register is saved.

T (optional) specifies that registers 14 and 15 are to be stored in words 4 and 5, respectively, of the save area. If both T and r2 are specified and r1 is any one of the registers 14, 15, 0, 1, or 2, all of registers 14 through r2 are saved.

name (optional) is an identifier name to be associated with the SAVE macro instruction. The name may be up to 70 characters and may be a complex name. If an asterisk is coded, the identifier is the symbol associated with the SAVE macro instruction, or, if the name field is blank, with the control section name. If the CSECT instruction name field is blank, the operand is ignored. Whenever a symbol or an asterisk is coded, the following macro expansion occurs:

(1) A count byte, containing the number of characters in the identifier name, is constructed, starting at four bytes following the address contained in GR15.
(2) The character string containing the identifier name is constructed, starting at five bytes following the address contained in GR15.

Description: The SAVE macro instruction causes the contents of the specified registers to be stored in the save area at the

address contained in GR13. An entry point identifier can optionally be specified. The SAVE macro should be written only at the entry point of a program because the code resulting from the macro expansion requires that GR15 contain the address of the SAVE macro instruction. See the "Calling Conventions" description in MTS Volume 3, System Subroutine Descriptions, for a further explanation of save areas and their formats.

Examples:

```
LAB1  SAVE   (14,12)
      USING  LAB1,15
```

This example generates a sequence to save registers 14 though 12 in the save area pointed to by register 13.

```
LAB2  SAVE   (5,10),T
      USING  LAB2,15
```

This example generates a sequence to save registers 5 through 10 in the save area pointed to by register 13. Registers 14 and 15 are stored in words 4 and 5 of the save area, respectively.

```
LAB3  SAVE   (14,12),,*
      USING  LAB3,15
```

This example generates a sequence to save registers 14 through 12 in the save area pointed to by register 13. The symbol LAB3 is associated with the SAVE macro instruction.

SCARDS

Macro Description

Purpose: To assemble a call to the SCARDS subroutine. See the SCARDS subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] SCARDS reg[,regl][,modifs][,lnr][,EXIT=exitseq] [,MF=mod][,TREG=(r1,r2)]

Parameters:

reg is the location of the region into which a record is to be read. This may be expressed as a symbol or as the number of a register (which contains the location of the region) in parentheses.

regl (optional) specifies the place to store the length of the input upon return from the SCARDS subroutine. This may be either the name of a halfword to contain the length, or the number of a register (which will contain the length) in parentheses. If omitted, the length is discarded.

modifs (optional) stands for several parameters separated by commas. Each parameter consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). If no modifiers are given, zero is used for the modifiers parameter to SCARDS. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.

lnr (optional) specifies the line number parameter for the SCARDS subroutine. This may be the location of a fullword containing the line number, a self-defining term which is the line number, or the number of a register (which contains the line number) in parentheses. If a register is specified, it will be stored in the line number parameter before the call and loaded from it after the call. If omitted, the macro generates a fullword if needed. If the @INDEXED modifier is specified in this macro or elsewhere, the value of the lnr parameter before the call to the SCARDS subroutine is used. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3 for further discus-

sion of this. If both regl and lnr are registers, lnr should not be GR1.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E. If omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. If modifs includes NOPROMPT and/or NOTIFY, the called subroutine will destroy the contents of register 0. If MF=(E,...) is specified, the contents of registers r1 and r2 are also destroyed. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced, and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, registers 14 and 15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call. If any modifiers are given, the new modifiers completely replace the old modifiers unless the MF parameter is given as MF=(E,listadr,SEP) in which case only the modifiers specified are changed.

Examples:

```
LAB1  SCARDS  REG,LEN,EXIT=EOF
```

This example calls SCARDS with REG as the location of the region to be read into and LEN as the location to store the length of the record read. A nonzero return code from SCARDS will cause a branch to EOF.

```
LAB2  SCARDS  REGION,LENG,@I,@PFX,EXIT=DONE
```

This example calls SCARDS with REGION as the location of the region to be read into and LENG as the location to store the length of the record read. The record is read with the @I and @PFX modifiers specified. A nonzero return code will cause a branch to DONE.

May 1983

SERCOM

Macro Description

Purpose: To assemble a call to the SERCOM subroutine. See the SERCOM subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype:

```
[label]  SERCOM  reg[,regl][,modifs][,lnr][,EXIT=exitseq]
                 [,MF=mod][,TREG=(r1,r2)]

[label]  SERCOM  'comment'[,modifs][,lnr][,EXIT=exitseq]
                 [,MF=mod][,TREG=(r1,r2)]
```

Parameters:

reg — is the location of the region from which the record is to be written. This may be expressed as a symbol or as the number of a register (which contains the location of the region) in parentheses.

regl — (optional) specifies the length of the output region for the SERCOM subroutine. This may be either the name of a halfword containing the length, or the number of a register (which contains the length) in parentheses. If it is omitted, L'REG is assumed.

modifs — (optional) stands for several parameters separated by commas. Each parameter consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). If no modifiers are given, zero is used for the modifiers parameter to SERCOM. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.

lnr — (optional) specifies the line number parameter for the SERCOM subroutine. This may be the location of a fullword containing the line number, a self-defining term which is the line number, or the number of a register (which contains the line number) in parentheses. If a register is specified, it will be stored in the line number parameter before the call and loaded from it after the call (if the @RETURNLINE# modifier is specified or if the MF=(E,...) form is used). If omitted, the macro generates a fullword if needed. If the @INDEXED modifier is

specified in this macro or elsewhere, the value of the lnr parameter before the call to the SERCOM subroutine is used. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3 for further discussion of this. If both regl and lnr are registers, lnr should not be GR1.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E. If omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. If modifs includes NOPROMPT and/or NOTIFY, the called subroutine will destroy the contents of register 0. If MF=(E,...) is specified, the contents of registers r1 and r2 are also destroyed. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced, and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, registers 14 and 15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call. If any modifiers are given, the new

modifiers completely replace the old modifiers unless the MF parameter is given as MF=(E,listadr,SEP) in which case only the modifiers specified are changed.

Examples:

```
LAB1  SERCOM  REG,LEN,EXIT=(EOF,ERROR)
```

This example calls SERCOM, writing the record contained in location REG of length contained in location LEN. A branch is made to EOF upon a return code of 4; a branch is made to ERROR upon a return code of 8 or greater.

```
LAB2  SERCOM  REGION,LENG,@I,EXIT=DONE
```

This example calls SERCOM, writing the record contained in location REGION of length contained in location LENG. The record is written with the @I modifier specified. A nonzero code from SERCOM will cause a branch to DONE.

```
LAB3  SERCOM  'THIS IS A COMMENT'
```

This example calls SERCOM, writing the text enclosed in primes.

SPIE

Macro Description

Purpose: To assemble a call to the SPIE subroutine. The SPIE subroutine is used to specify the address of an interruption exit routine and to specify the program interrupt types that are to cause the exit routine to be given control. See the SPIE subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] SPIE [exit,(mask)][,MF=mod]

Parameters:

exit is the location of the exit routine to be given control after a program interrupt of the type specified in the mask parameter.

mask is one or more decimal numbers, separated by commas, specifying the program interrupt types which cause control to be given to the exit routine. These decimal numbers correspond to the 15 program exception codes. See the SPIE subroutine description in MTS Volume 3. The interrupt types can be specified in any order as follows:

(1) One or more single numbers, each indicating the program interrupt type.
(2) One or more pairs or numbers, each pair indicating a range of interrupt types. The second number must be higher than the first. The pair of numbers must be separated by commas and enclosed in parentheses.

mod is a keyword parameter. See below for a description of the MF= keyword.

This macro destroys the contents of registers 1, 14, and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced and all other parameters are optional.

If MF=(E,listadr) is specified, listadr is assumed to be the address of a remote parameter list, and only the

executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the remote parameter list.

Examples:

```
SPIE   FIXUP,(8)
```

This example calls SPIE with FIXUP as the exit routine and the number 8 (fixed-point overflow) as the interrupt mask.

```
SPIE   FIXUP,((8,15))
```

This example calls SPIE with FIXUP as the exit routine and the range 8 through 15 as the interrupt mask.

```
SPIE   MF=(E,(5))
```

This example resets the SPIE exits to those specified by the address contained in register 5. This could be a value returned in register 1 by a previous SPIE.

SPRINT

Macro Description

Purpose: To assemble a call to the SPRINT subroutine. See the SPRINT subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype:

```
[label]  SPRINT  reg[,regl][,modifs][,lnr][,EXIT=exitseq]
                 [,MF=mod][,TREG=(r1,r2)]

[label]  SPRINT  'comment'[,modifs][,lnr][,EXIT=exitseq]
                 [,MF=mod][,TREG=(r1,r2)]
```

Parameters:

reg is the location of the region from which the record is to be written. This may be expressed as a symbol or as the number of a register (which contains the location of the region) in parentheses.

regl (optional) specifies the length of the output region for the SPRINT subroutine. This may be either the name of a halfword containing the length, or the number of a register (which contains the length) in parentheses. If it is omitted, L'REG is assumed.

modifs (optional) stands for several parameters separated by commas. Each parameter consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). If no modifiers are given, zero is used for the modifiers parameter to SPRINT. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.

lnr (optional) specifies the line number parameter for the SPRINT subroutine. This may be the location of a fullword containing the line number, a self-defining term which is the line number, or the number of a register (which contains the line number) in parentheses. If a register is specified, it will be stored in the line number parameter before the call and loaded from it after the call (if the @RETURNLINE# modifier is specified or if the MF=(E,...) form is used). If omitted, the macro generates a fullword if needed. If the @INDEXED modifier is

specified in this macro or elsewhere, the value of the lnr parameter before the call to the SPRINT subroutine is used. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3 for further discussion of this. If both regl and lnr are registers, lnr should not be GR1.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E. If omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. If modifs includes NOPROMPT and/or NOTIFY, the called subroutine will destroy the contents of register 0. If MF=(E,...) is specified, the contents of registers r1 and r2 are also destroyed. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced, and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, registers 14 and 15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call. If any modifiers are given, the new

modifiers completely replace the old modifiers unless the MF parameter is given as MF=(E,listadr,SEP) in which case only the modifiers specified are changed.

Examples:

```
LAB1  SPRINT  REG,LEN,EXIT=(EOF,ERROR)
```

This example calls SPRINT, writing the record contained in location REG of length contained in location LEN. A branch is made to EOF upon a return code of 4; a branch is made to ERROR upon a return code of 8 or greater.

```
LAB2  SPRINT  REGION,LENG,@I,EXIT=DONE
```

This example calls SPRINT, writing the record contained in location REGION of length contained in location LENG. The record is written with the @I modifier specified. A nonzero code from SPRINT will cause a branch to DONE.

```
LAB3  SPRINT  'THIS IS A COMMENT'
```

This example calls SPRINT, writing the text enclosed in primes.

SPUNCH

Macro Description

Purpose: To assemble a call to the SPUNCH subroutine. See the SPUNCH subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype:

```
[label]  SPUNCH  reg[,regl][,modifs][,lnr][,EXIT=exitseq]
                 [,MF=mod][,TREG=(r1,r2)]

[label]  SPUNCH  'comment'[,modifs][,lnr][,EXIT=exitseq]
                 [,MF=mod][,TREG=(r1,r2)]
```

Parameters:

- reg — is the location of the region from which the record is to be written. This may be expressed as a symbol or as the number of a register (which contains the location of the region) in parentheses.
- regl — (optional) specifies the length of the output region for the SPUNCH subroutine. This may be either the name of a halfword containing the length, or the number of a register (which contains the length) in parentheses. If it is omitted, L'REG is assumed.
- modifs — (optional) stands for several parameters separated by commas. Each parameter consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). If no modifiers are given, zero is used for the modifiers parameter to SPUNCH. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.
- lnr — (optional) specifies the line number parameter for the SPUNCH subroutine. This may be the location of a fullword containing the line number, a self-defining term which is the line number, or the number of a register (which contains the line number) in parentheses. If a register is specified, it will be stored in the line number parameter before the call and loaded from it after the call (if the @RETURNLINE# modifier is specified or if the MF=(E,...) form is used). If omitted, the macro generates a fullword if needed. If the @INDEXED modifier is

specified in this macro or elsewhere, the value of the lnr parameter before the call to the SPUNCH subroutine is used. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3 for further discussion of this. If both regl and lnr are registers, lnr should not be GR1.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E. If omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. If modifs includes NOPROMPT and/or NOTIFY, the called subroutine will destroy the contents of register 0. If MF=(E,...) is specified, the contents of registers r1 and r2 are also destroyed. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced, and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, registers 14 and 15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call. If any modifiers are given, the new

modifiers completely replace the old modifiers unless the MF parameter is given as MF=(E,listadr,SEP) in which case only the modifiers specified are changed.

Examples:

```
LAB1  SPUNCH  REG,LEN,EXIT=(EOF,ERROR)
```

This example calls SPUNCH, writing the record contained in location REG of length contained in location LEN. A branch is made to EOF upon a return code of 4; a branch is made to ERROR upon a return code of 8 or greater.

```
LAB2  SPUNCH  REGION,LENG,@I,EXIT=DONE
```

This example calls SPUNCH, writing the record contained in location REGION of length contained in location LENG. The record is written with the @I modifier specified. A nonzero code from SPUNCH will cause a branch to DONE.

```
LAB3  SPUNCH  'THIS IS A COMMENT'
```

This example calls SPUNCH, writing the text enclosed in primes.

STIMER

Macro Description

Purpose: MTS support for the OS STIMER macro.

Prototype: [label] STIMER type,[exit],units=time

Parameters:

type is one of the following:

TASK specifies that the time interval is to decremented only when the associated task is running.

REAL specifies that the time interval is to be decremented continuously.

WAIT specifies that the time interval is to be decremented continuously and that the task is to be placed in wait state until the interval has elapsed.

exit (optional) specifies the address of an exit routine to be called when the interval expires. If not specified, no routine is called.

units is one of the following:

TUINTVL time is an unsigned 32-bit binary integer giving the time interval in "OS timer units", where one OS timer unit is 26 1/24 microseconds. Note that an OS timer unit does not match any actual hardware timer unit.

BINTVL time is an unsigned 32-bit binary integer giving the time interval in hundredths of a second.

DINTVL time is an 8-byte character string giving the time interval in the form HHMMSSTH, where HH is hours, MM is minutes, SS is seconds, T is tenths of a second, and H is hundredths of a second.

TOD time is an 8-byte character string giving the time of day at which the interval is to expire, in the form HHMMSSTH (see above). If TASK is specified, TOD is equivalent to DINTVL.

time is the location of the 4- or 8-byte fullword-aligned quantity described by the units parameter.

This macro destroys the contents of registers 14 and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: This macro generates a call to the STIMER subroutine, located in *LIBRARY.

When it is executed, a timer interrupt is set up, which will expire after the specified time interval. When the interval expires, the subroutine exit will be called, if the exit parameter is specified. At the time of this call, the registers will be as follows:

GR1 = address of an exit region containing the following information.

bytes 0-7: PSW at time of interrupt
bytes 8-71: GR0-GR15 at time of interrupt

GR13 = address of a 72-byte save area
GR14 = return address
GR15 = address of exit subroutine entry

FPRS unchanged from time of interrupt

If the exit subroutine returns, the program will be restarted from the point of the interrupt.

There can be at most one STIMER interval set up at a time. If a second STIMER macro call is executed before the first has expired, the second call overrides the first call.

The STIMER macro and the corresponding subroutine are provided primarily for programs converted from IBM System/360 Operating System (OS) which use the OS STIMER macro.

MTS users are advised to use the SETIME, TIMNTRP, RSTIME, GETIME, TWAIT, or TICALL subroutines, described in MTS Volume 3, System Subroutine Descriptions.

The parameters for the STIMER subroutine, as generated by the macro, are:

GR0 - bits 0-3: 0=TUINTVL
1=BINTVL
3=DINTVL
7=TOD

```
                bits 4-7:  0=TASK
                           1=WAIT
                           3=REAL
                bits 8-31: exit address or zero
          GR1 - address of time interval, as described by GR0,
                bits 0-3.
```

See the TTIMER macro description in this volume for further information.

Example:

```
        STIMER WAIT,,TOD=TWO30
        STIMER TASK,EXIT,BINTVL=TENSEC
         .
         .
EXIT    STM    14,12,12(13)
         .
         .
TWO30   DC     C'02300000'
TENSEC  DC     F'1000'
```

SYSTEM

Macro Description

Purpose: To assemble a transfer to the SYSTEM subroutine. See the SYSTEM subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype: [label] SYSTEM

Description: This will generate

```
label  L    15,=V(SYSTEM)
       BR   15
```

TRL, TRTL

Macro Description

Purpose: To provide a long form of the Translate and Translate and Test machine instructions similar to the MVCL and CLCL formats.

Prototype:
```
[label]  TRL   r1,d2(b2)
[label]  TRTL  r1,d2(b2)
```

Parameters:

- r1 is an even register specifying an even-odd register pair giving the location and length (respectively) of the operand to be translated or tested.
- d2(b2) is the location of the translate or test table just as it would be for the TR or TRT instructions.

Description: Instructions are generated which perform the translate or translate and test machine instructions on operands of any length. The operation proceeds just as described in the IBM Principles of Operation manuals for TR and TRT instructions.

Only bits 8-31 of r1 and r1+1 are used in the operation. The contents of r1 and r1+1 are undefined when the operation is terminated. The condition code for TRL is unpredictable. The condition code for TRTL is similar to that of the TRT instruction.

TRTL should not use register pairs 0-1 or 2-3 for the first operand, since registers 1 and 2 are potentially changed by the macro.

TRTAB

Macro Description

Purpose: To set up a translate or translate-and-test table.

Prototype: [label] TRTAB item,item,...[,SIZE=size][,START=offset]
[,RULER=rule][,FILL=filler][,BASE=filler]

Parameters:

item is one of the following:

(1) BEGIN
(2) END
(3) value
(4) (value1,value2,...,insert)

Form (1) is used to specify a series of TRTAB macros for a single table. "BEGIN" should be the first item of the TRTAB macro; TRTAB macros may be issued until the item "END" is encountered. If the first item is not "BEGIN" and the table has not been initialized by a previous TRTAB macro whose first item was "BEGIN", the table is automatically initialized and the code is generated after the last item has been processed. If "BEGIN" is specified, the parameters SIZE, START, BASE, and FILL will be ignored in the TRTAB macros other than the first, until "END" is encountered.

Form (2) generates the code of the table. It is required when the first operand of a previous or the current TRTAB macro is "BEGIN". Any remaining items after "END" in the TRTAB macro are ignored.

Form (3) indicates that the "value" is placed at the offset "value" from the label of the table. For example, 'ABC' places 'A', 'B', 'C' at the offsets 'A', 'B', 'C' of the table, respectively.

"value" can be a one-byte or a multibyte string. A one-byte value is any valid absolute integer expression, with all terms already defined. Examples are: X'FA', C'?', 64, C'0'-C'A'. Multibyte values are

processed one byte at a time from left to right. They are of two forms: character string ('string') or hexadecimal string (X'hex' or 'hex'X). Each character in 'string' can be any valid EBCDIC character with every prime and ampersand doubled; '''' is same as C'''', 'XYZ' same as C'X', C'Y', C'Z'. Each byte in "hex" must consist of two valid hexadecimal digits (0-9, A-F). For example, '00FF'X or X'00FF' is same as X'00', X'FF'.

Optionally, "value" may be of the form "value1...value2", that is, the two values are separated by an ellipsis. This is interpreted as going from "value1" through "value2", inclusively. Thus, 'A'...'F' is same as 'A','B','C','D','E','F'. The direction may be backwards, e.g., 'F'...'A' is same as 'FEDCBA'. Both "value1" and "value2" should be within (0,255), inclusively. The old form "value1-value2", where values are separated by a minus-sign "-" and "value1" is less than "value2", will be interpreted as the equivalent of "value1...value2".

Form (4), enclosed within parentheses, is used most often. First, "insert" is generated into a list of one-byte replacement values. The list "insert" may be a sublist enclosed within parentheses. For example, ('A','B','C') for insert is equivalent to 'ABC'. Alternatively, "insert" may be expressed of form "*+disp" or "*-disp", where "disp" is any valid absolute integer expression. Examples are "*", "*+1", "*-C'0'". For compatibility with the old form, "+disp" and "-disp" will be interpreted as "*+disp" and "*-disp", respectively. The displacement is added to (or subtracted from) the relative table offset of each byte it replaces. For example, to put the character equivalent of the digits in their own relative table locations,

```
TRTAB ('0'...'9',*)
```

and to replace each letter by the next higher letter, wrapping around from Z to A,

```
TRTAB ('A'...'H',*+1),                +
('J'...'Q',*+1),                      +
```

```
('S'...'Y',*+1),                    +
('IRZ','JSA')
```

If the list of replacement values is exhausted before the values are exhausted, the first value will be reused, and then the second, etc. This wraparound feature can be used to place the same replacement value in all offsets, e.g., ('ABCDEF',8) places 8 at offsets 'A' through 'F' in the table.

size — specifies the number of bytes of translate table to generate. It defaults to 256.

offset — is the offset of the table generated from the label of the table. It defaults to 0. "offset" can be any valid one-byte absolute assembly expression, character or hexadecimal string (e.g., 240, '0', X'F0', or 'F0'X).

rule — If YES, a ruler will be written around the generated code of this table. This may be overridden by RULER=NO. The default is YES. This parameter remains in effect for all remaining TRTAB macros until it is overridden by the RULER parameter of another TRTAB macro.

filler — specifies what value is generated for each table entry not specified by the items. It defaults to 0. This may be any absolute integer expression (e.g., X'FA', C'.', or 12) which is evaluated within (0,255), inclusively. A one-byte character string (e.g., '?') or a one-byte hexadecimal string (e.g., '11'X) is allowed. An alternate form FILL=* fills the entire table with values X'00' through X'FF' and can be used for translating.

Examples: This example generates a translate table for hexadecimal output:

```
HEXTRA  TRTAB  (X'FA'...X'FF','ABCDEF'),          +
               FILL=*,START='0',SIZE=16
```

This is same as:

```
HEXTRA  EQU    *-240,256,C'X'
        DC     C'0123456789ABCDEF'
```

The following example marks with 0 all characters legal in any floating-point number and the remaining characters with 4.

```
     FLTABLE TRTAB  ('+-.E','0'...'9',0),FILL=4
```

The following table is used to translate all uppercase letters to lowercase letters.

```
     LCTABLE TRTAB  ('ABCDEFGHIJKLMNOPQRSTUVWXYZ',      +
                    'abcdefghijklmnopqrstuvwxyz'),      +
                    FILL=*
```

The following table translates all unprintable characters to '?'. Note the use of "BEGIN" and "END".

```
     PRNTBL  TRTAB  BEGIN,FILL='?'
             TRTAB  '0123456789'
             TRTAB  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
             TRTAB  'abcdefghijklmnopqrstuvwxyz'
             TRTAB  '=<;:%''>*()_+-&&¢@!"$#|¬?,./'
             TRTAB  END
```

May 1983

TTIMER

Macro Description

Purpose: MTS support for the OS TTIMER macro.

Prototype: [label] TTIMER [CANCEL]

Parameters:

CANCEL if specified, the currently set up timer interval will be cancelled; otherwise, it will not be cancelled.

This macro destroys the contents of registers 14 and 15. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: The TTIMER macro generates a call to the TTIMER subroutine, located in *LIBRARY.

The TTIMER macro returns, in GR0, the time remaining in the time interval set up by the STIMER macro. If no such time interval is currently active, GR0 will contain zero. The remaining time is returned in OS timer units (an unsigned 32-bit integer; one OS timer unit is 26 1/24 microseconds).

The TTIMER macro and the corresponding subroutine are provided primarily for programs converted from IBM System/360 Operating System (OS) which use the OS TTIMER macro.

MTS users are advised to use the SETIME, TIMNTRP, RSTIME, GETIME, TWAIT, or TICALL subroutines, described in MTS Volume 3, System Subroutine Descriptions.

The parameters for the TTIMER subroutine, as generated by the macro, are:

```
GR1 - 0 -> no cancel
      1 -> cancel
```

See the STIMER macro description in this volume for further details.

Example:

```
TTIMER
TTIMER CANCEL
```

WRITE

Macro Description

Purpose: To assemble a call to the WRITE subroutine. See the WRITE subroutine description in MTS Volume 3, System Subroutine Descriptions.

Prototype:

```
[label]  WRITE   unit,reg[,regl][,modifs][,lnr]
                 [,EXIT=exitseq][,MF=mod][,TREG=(r1,r2)]

[label]  WRITE   unit,'comment'[,modifs][,lnr]
                 [,EXIT=exitseq][,MF=mod][,TREG=(r1,r2)]
```

Parameters:

- unit — specifies the corresponding parameter to be given to the WRITE subroutine. This parameter is either a number from 0 through 19, the name of a logical I/O unit enclosed in primes, the location of a fullword-aligned fullword containing a FDUB-pointer, or the number of a register (which contains a FDUB-pointer) in parentheses.
- reg — is the location of the region from which the record is to be written. This may be expressed as a symbol or as the number of a register (which contains the location of the region) in parentheses.
- regl — (optional) specifies the length of the output region for the WRITE subroutine. This may be either the name of a halfword containing the length, or the number of a register (which contains the length) in parentheses. If it is omitted, L'REG is assumed.
- modifs — (optional) stands for several parameters separated by commas. Each parameter consists of the name of an MTS modifier preceded by either an at sign (@), an at sign and a not sign (@¬), or an at sign and a minus sign (@-). If no modifiers are given, zero is used for the modifiers parameter to WRITE. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3.
- lnr — (optional) specifies the line number parameter for the WRITE subroutine. This may be the location of a fullword containing the line number, a self-defining term which is the line number, or the number of a

register(which contains the line number) in parentheses. If a register is specified, it will be stored in the line number parameter before the call and loaded from it after the call (if the @RETURNLINE# modifier is specified or if the MF=(E,...) form is used). If omitted, the macro generates a fullword if needed. If the @INDEXED modifier is specified in this macro or elsewhere, the value of the lnr parameter before the call to the WRITE subroutine is used. See the "I/O Modifiers" description in MTS Volume 1 or MTS Volume 3 for further discussion of this. If both regl and lnr are registers, lnr should not be GR1.

exitseq (optional) is a keyword parameter specifying the exits to be taken for nonzero return codes. If exitseq is a single symbol, any nonzero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, etc. If a return code larger than that corresponding to the last symbol occurs, a branch to the last symbol will be taken.

mod (optional) is a keyword parameter. See below for a description of the MF= keyword.

r1,r2 (optional) is a keyword parameter specifying two temporary registers to be used in expanding calls when MF=E. If omitted, GR14 and GR15 are assumed.

This macro destroys the contents of registers 1, 14, and 15. If modifs includes NOPROMPT and/or NOTIFY, the called subroutine will destroy the contents of register 0. If MF=(E,...) is specified, the contents of registers r1 and r2 are also destroyed. Register 13 must point to the calling program's save area. The condition code may be changed.

Description: If MF=L is included in the parameter list, only the parameter list will be generated. In this case, no executable code is produced, and all other parameters are optional. Parameters specified as registers will be ignored.

If MF=(E,listadr) is specified, listadr is assumed to be the name assigned to an MF=L form of the macro, and only the executable code required to call the subroutine is generated. listadr may be the name of an MF=L form of the macro or the number of a register (which contains the

location of an MF=L form) in parentheses. If any other parameters are given, they are used to modify the list generated by the MF=L macro before the call. In this case, the TREG keyword specifies two registers to be used for modifying the parameter list. If omitted, registers 14 and 15 are used, and they should not contain parameters. Any other parameter may be used with an MF=E form of the call. If any modifiers are given, the new modifiers completely replace the old modifiers unless the MF parameter is given as MF=(E,listadr,SEP), in which case only the modifiers specified are changed.

Examples:

```
LAB1  WRITE   6,REG,LEN,EXIT=(EOF,OUCH)
```

This example calls WRITE, specifying that a record is to be written to logical I/O unit 6 from the region at location REG of length specified by location LEN. A branch is made to EOF upon a return code of 4 from WRITE; a branch is made to OUCH upon a return code of 8 or greater.

```
LAB2  WRITE   'SPRINT',REGION,LENG,@I,EXIT=DONE
```

This example calls WRITE, specifying that a record is to be written to SPRINT from the region at location REGION of length specified by location LENG. The record is written with the @I modifier specified. A branch is made to DONE upon a nonzero return code from WRITE.

```
LAB3  WRITE   7,'THIS IS A COMMENT'
```

This example calls WRITE, specifying that the text enclosed in primes is to be written to logical I/O unit 7.

May 1983

THE MESSAGE MACROS

This section describes a group of three macros which assist the 360/370-assembler language programmer in the construction and output of messages. These three MSG macros, PHRASE, MSG, and PMSG, accept identical operators. PHRASE assembles a partial (unterminated) message, while MSG and PMSG assemble complete messages.

The operators to the macros are translated into sequences of message items which are translated into message text at execution time by the resident MSG routine. The PMSG macro differs from the MSG macro only in that it invokes that routine to print the message. As each item is processed, the MSG routine may act on an internal buffer (the message buffer) in some way. The most common operation is the addition of text to the buffer, but other items can cause the buffer to be printed or modify the position at which the next addition will be placed.

There is a limit of 255 characters per output line in a message.

EXAMPLES

(1) PHRASE Macro

```
PHRASE 'Hello '
PHRASE (NAME),'.'
PHRASE END
```

This trivial example shows a complete message built of a series of PHRASE's. There can be several operators in one PHRASE statement. Notice that the message is eventually terminated by the END operator.

(2) MSG macro

```
MSG 'Hello ',(NAME),'.'
```

This example shows the same complete message. In this case, the terminating END is not given since it is generated by the MSG macro.

(3) PMSG Macro

```
PMSG 'Hello ',(NAME),'.'
```

This example still assembles the same complete message, but also prints it.

```
              PMSG HELLOMSG
                .
                .
                .
     HELLOMSG MSG 'Hello ',(NAME),'.'
```

This is another way of saying the same thing.

MSG MACRO OPERATORS

Messages consist of a sequence of message-items. Each message-item is an operator followed by zero or more operands. Operands which refer to addresses in storage generally use S-type address constants for these references. The S-type constants are resolved by reference to the registers in use when the MSG routine was called. The CALLER operator may be used to back up to the previous savearea and thus use the registers of the program that called the caller of MSG. In cases where an S-type address constant is used, the special case with base register 0 and displacement less than or equal to 15 is taken to mean that the quantity to be processed is contained in the register equal to the displacement.

Messages may be nested either statically (by means of BEGIN/END blocks) or dynamically (by means of the MESSAGE operator). An entire message is allowed anywhere that a single message-item can appear (e.g., following IF(...)).

The overall syntax of legal messages is as follows.

```
  <message>   ::= <phrase> ... END
  <phrase>    ::= BEGIN <phrase> ... END
                | LOOP <phrase> ... END
                | IF(...) <phrase>
                | IF(...) <phrase> ELSE <phrase>
                | WIDTH(...) <phrase>
                | PACKET <phrase>
                | <other operator>
```

The following operators are recognized by this version of MSG. Any text which is not preceded by a valid operator is treated as text to be inserted in the message.

In the descriptions below, where LEN is specified, BLEN, HLEN, or FLEN may be specified to explicitly indicate a relocatable byte, halfword or fullword length. If LEN is used with a relocatable expression, the MSG routine accepts a length stored in either a halfword or a fullword.

In the following, "re" means "relocatable expression" (i.e., an S-type address constant), and "nre" means "nonrelocatable expression" (i.e., a constant). In the following macro prototypes, keywords (xxx=...) indicate optional items.

MSG Operator	Explanation
/ or PRINT	Print the message. Prints the current contents of the message buffer to all output routes currently requested, and empties the buffer. Examples: 'Hi There.',/ 'It''s me!!',PRINT
'xxxx'	Literal string. The text of the literal is inserted into the message. Example: 'This is a message.'
(name[,LEN={re|nre}][,{END|DELIM}=nre])	Generalized insertion by length and/or delimiter. This operator provides generalized insertion in which either a length and/or a delimiter may be given. If both are given, then the specified number of bytes or up to the delimiter, whichever comes first, are moved to the buffer. The length may be given as either an "nre" or "re"; if "re" is given, the length may be either a halfword or a fullword. If both the length and delimiter field are omitted, L'name is assumed. Examples: (X,LEN=Y,DELIM=C' ') (X,LEN=Y) (X,DELIM=X'FF') (X,LEN=6) (X)

name — Nested message name.

Nested messages are included where their invocations are encountered. Note that a nested message behaves as an inner block. (See the BEGIN and END operator descriptions below.)

If the value of the message name is a number less than or equal to 15, then it is assumed to be the number of a register which contains the location of the message.

Examples:

```
MSG2
R2
```

ADDR(name[,LEN={nre|re}]) — Hexadecimal output conversion.

This is the same as HEX, except that left-zeros are suppressed.

Examples:

```
ADDR(X,LEN=3)
ADDR(X,LEN=XLEN)
```

BEGIN — Begins a message block.

A BEGIN...END group defines the limits of a message block. They are mainly used to enable a false IF to skip an entire group of message operations at once, or to cause a WIDTH item to process a group of message items. A common construction is:

```
IF(...),BEGIN,...,END
```

The current fill character is saved by BEGIN and restored by END.

Example:

```
BEGIN,FILL(C'0'),WIDTH(4),DEC(RC),END
```

CALL(re¹,re²,...,re⁶) — Subroutine call.

The subroutine indicated by the relocatable expression re¹ is called with a standard OS S-type calling sequence. The re² up to re⁶ (maximum) are relocatable expressions passed to the subroutine as the second up to sixth arguments. The first argument is the location

of a vector of three fullwords containing the location of the end of the output buffer, the location of the next character in the output buffer, and the location of the beginning of the output buffer, respectively. Also, at the time of the call, register 0 contains the location of a null message.

On return, the internal condition code (which may be tested by the IF operator) is set to bits 24-31 of register 15 unless the value in register 15 is negative, in which case the internal condition code is not changed. Also, it is assumed that the subroutine has placed into register 0 the location of a message to be nested in the current message. Note that if the subroutine restores the register 0 supplied on the call, the null message will be used.

Example:

```
CALL(SUBR,P1,P2)
```

CALLER(name) Changing register references for a nested message.

The nested message, name, will be expanded with all register-references resolved from the previous savearea. The caller of MSG must have been called with a standard OS calling sequence.

This operator is used when an intermediary routine exists between the MSG routine and the message emitting routine.

Examples:

```
CALLER(MESS)
CALLER(R4)
```

DEC(name[,LEN={nre|re}]) Decimal output conversion.

The number starting at the location name is converted to decimal and placed in the output line. Leading zeros are omitted. LEN specifies the number of bytes in the number, it defaults to L'name and must not exceed 4.

If LEN is even, the number is treated as a signed integer; otherwise, it is treated as unsigned.

Examples:

```
DEC(N,LEN=2)
DEC(NUM)
DEC(N,LEN=Z)
DEC(R2,LEN=4)
```

END — Ends the current BEGIN block, LOOP loop, or the entire message.

Note that the END operator is what terminates every message. An END is automatically supplied by the MSG and PMSG (but not PHRASE) macros. This automatic END is the only difference between the MSG and the PHRASE Macros.

Example:

```
BEGIN,...,END
```

ELSE — The ELSE operator combines with a preceding IF (any of the three varieties). If the message following the IF was skipped, then the message following the ELSE will be included, and vice versa.

Example:

```
IF(X,EQ,Y),'True',ELSE,'False'
```

EXIT — Exits the closest enclosing LOOP loop.

Example:

```
LOOP,...,IF(...),EXIT,END
```

FILL(nre) — Setting the fill character.

This sets the fill character, used in tabbing or width padding, to nre. The default fill character is a blank.

Example:

```
FILL(C'*')
```

GOTO(re) — Transfer control.

The message scan continues from the address given as re which should be the label on a MSG or PHRASE macro. Note that it does not detect branching out of a block, etc. It is the

user's responsibility to ensure the message nesting level is correct.

Example:

```
GOTO(MSG2)
```

HEX(name[,LEN={nre|re}]) Hexadecimal output conversion.

The value starting at location name is converted to hexadecimal output form and placed in the output line. LEN specifies the number of bytes to be converted and may be either nonrelocatable or relocatable. LEN defaults to L'name.

Examples:

```
HEX(X,LEN=10)
HEX(A)
```

IF(re,op,nre) Conditional (one-byte compare).

The byte specified by re is compared (as specified by op) with the comparand nre. Complete details are given in the section on IF operators.

Example:

```
IF(X,EQ,5),MSG
```

IF(re¹,op,re²[,LEN={nre|re}])

Generalized conditional.

This is a generalization of IF that allows a length to be specified. The two operators are relocatable symbols, and the length is specified as an nre or re. Complete details are given in the section on IF operators.

Example:

```
IF(X,EQ,Y,LEN=100),MSG
```

IF(*,op,nre) Conditional (internal condition code.)

This form is used to test the internal condition code, for example, as set by the CALL operator. Complete details are given in the section on IF operators.

Example:

```
IF(*,EQ,4),MSG
```

LOOP — The following message items, down to the matching END, are processed repeatedly until the loop is terminated with an EXIT operator.

Example:

```
LOOP,...,END
```

LNR(name[,LEN={nre|re}]) — Line-number conversion.

This is the same as the DEC operator, except name is assumed to have the internal form of an MTS line number. The value is divided by 1000 and printed with a decimal point with up to three digits after the decimal point. Leading zeros, and trailing zeros after the decimal point are omitted.

Example:

```
LNR(N)
```

OUTPUT(...) — Set output routing.

Specifies the place or places to which subsequent output is to be directed. Complete details are given in the section on OUTPUT routing.

Note that several routes may be specified with one OUTPUT operator.

Example:

```
OUTPUT(SERCOM,CONTROL(FDUB=MSINKFDUB))
```

PACKET — The PACKET operator specifies that the message buffer is composed of a one-byte field giving the message length followed by the message text.

Examples:

```
PACKET,'HELLO'
PACKET,BEGIN,...,END
```

PICTURE(name[,LEN={nre|re}],'pattern')
Decimal conversion with "picture" format.

This operator performs decimal conversion of the binary value given by name with length LEN. The picture specifies the resulting output format. It is a PL/I-like picture specification, more fully discussed in the PICTURE operator section.

Example:

```
PICTURE(X,LEN=2,'zzz9.zzz')
```

RETURN(nre) Returning from a message.

The current message buffer is written and the message routine returns to the caller with a return code equal to nre.

Note this causes an immediate exit, regardless of the message nesting depth.

Examples:

```
RETURN(4)
RETURN(A-B)
```

SKIP(nre) Moving buffer pointer.

The output buffer pointer is moved by the specified number of positions to the left or right of its current position. If nre is negative, the movement is to the left; if nre is positive, the movement is to the right. The maximum value for nre is 127. When "skipping" to the right, the intervening spaces are filled with the current fill character.

Example:

```
SKIP(-5)
```

SYMBOL(name[,LEN={nre|re}])
Address conversion to character string.

If loader information is available, the address starting at location name with length LEN is printed in the form:

```
symbol+displacement
```

Otherwise, the address is treated the same as ADDR.

Example:

```
SYMBOL(Y)
```

TAB(nre) — Tabbing.

The output buffer pointer is moved to the column given by nre. The first column has column number 1. TAB(0) is a no-op. Tabbing to the right causes intervening spaces to be filled with the current fill character. Tabbing left is allowed but does not start a new output line. Note: COL is a synonym for TAB.

Example:

```
TAB(20)
```

WIDTH(nre,posn) — Specifying the width of a message item.

This causes the next message item (either single item, block or embedded message) to be placed in a fixed-width field of nre columns, positioned in that field as specified by the second operator. The padding is done with the current fill character. The default positioning is right-justified.

posn may be LEFT, RIGHT, CENTRE, or CENTER.

Example:

```
WIDTH(20,RIGHT),MSG
```

THE "IF" OPERATORS

The IF operator causes the next message item or message block (see BEGIN operator) to be included if the condition is true; otherwise, it will be skipped. If the message item being skipped is another IF operator, then that and any other IF's immediately following, are skipped up to and including the first message item or block that is not an IF.

IF Operator Formats

The IF operator exists with three different but similar formats. In the examples below, re is a relocatable expression while nre is a nonrelocatable expression. op specifies the comparison operator.

The IF operators are:

(1) IF(re,op,nre) or IF(re)

This version of the IF operator compares a one-byte field re to nre. In the case of the single field, it is assumed that re is a flag defined by the FLAGS macro.

```
Examples:  IF(SWS,EQ,X'08'),'Switch flag = 08'
           'Batch Flag is ',IF(BATCH),'on',ELSE,'off'
```

(2) IF(*,op,nre)

The one-byte internal condition code of MSG (indicated by the "*") is to be compared with nre.

Note that the internal condition code is set by the CALL operator return code and by various I/O routes designated by the OUTPUT operator.

```
Example:   IF(*,EQ,4),'Internal Code = 4'
```

(3) IF(re¹,op,re²,LEN={nre|re})

This is a generalized version of the IF operator and compares re¹ to re². LEN specifies the length of the compare to be used. If it is omitted, L're¹ is used.

```
Example:   IF(ORL,ALT,F55,LEN=4),SHORTMSG,ELSE,LONGMSG
```

IF Comparison Operators

An IF comparison operator indicates two things to the MSG routine: the comparison "type" and the branch condition for the operation.

There are three types of comparison operators (flagged by op in the above IF prototype statements).

(1) "Compare Logical" comparison

The "compare logical" comparison operators are: EQ, NE, GT, GE, LT, and LE (they use the "CLC" machine instruction).

(2) "Test under Mask" comparison

The "test under mask" comparison operators are: OA, ZA, MA, NA or OT, ZT, MT, NT (they use the "TM" machine instruction).

(To help understand the meaning of these mnemonics, OA means "ones when anded with," while OT means "ones when tested with." Note these are equivalent.)

(3) "Arithmetic Compare" comparison

The "arithmetic compare" operators are: AEQ, ANE, AGT, AGE, ALT, and ALE (MSG uses a "CR" machine instruction to perform the comparison).

OUTPUT OPERATOR

The OUTPUT operator[1] specifies the place or places to which subsequent output is to be directed.

It should be noted that if the OUTPUT operator is omitted from a message, then, by default, the message will be routed through SERCOM.

In the following lists of OUTPUT operators, parameters in square brackets "[]" are optional and may be omitted. The optional parameters are discussed later.

OUTPUT operators

(1) OUTPUT(SERCOM[(MODS=nre,LNR=re,ORL=re)])

Write message to SERCOM.

(2) OUTPUT(SPRINT[(MODS=nre,LNR=re,ORL=re)])

Write message to SPRINT.

(3) OUTPUT(SPUNCH[(MODS=nre,LNR=re,ORL=re)])

Write message to SPUNCH.

---

[1]This operator replaces the OPTIONS operator from MSG (Version I). The OPTIONS operator is obsolete and should not be used any more.

(4) OUTPUT(WRITE(FDUB=re[,MODS=nre,LNR=re,ORL=re]))

Write message using the FDUB-pointer at the location specified by FDUB=re.

(5) OUTPUT(SUBROUTINE(ADDR=re[,FDUB=re,MODS=nre,LNR=re,ORL=re]))

The location specified by ADDR=re contains the address of a subroutine with 'WRITE-like' parameters that is to be used to dispose of the message.

(6) OUTPUT(OPER)

Write message to the operator's console.

(7) OUTPUT(CMD)

Call the CMD subroutine with the output line.

(8) OUTPUT(MTSCMD)

Call the MTSCMD subroutine with the output line.

(9) OUTPUT(CMDNOE)

Call the CMDNOE subroutine with the output line.

(10) OUTPUT(CONTROL(FDUB=re))

Call the CONTROL subroutine using the fdub pointer from FDUB=re.

(11) OUTPUT(MEMORY(ADDR=re[,ORL=re]))

Store the resulting output line at the address specified by ADDR=re. (Note that the PACKET operator can be used to cause the message length to be stored in the byte at the head of the message.)

Optional OUTPUT Operators

The following optional operators are only allowed with certain of the OUTPUT operators, as illustrated in the above descriptions.

(1) MODS=nre

This operator specifies the modifier bits for the I/O routines. It is a nonrelocatable expression which may be either a hexadecimal string or it may be a parenthesized expression suitable for the MTSMODS macro.

Examples: ...(MODS=40000040) or ...(MODS=(@¬CC,@ERRRTN))

(2) LNR=re

This operator specifies the location of a linenumber to be used by the I/O routines.

(3) ORL=re

This operator specifies a halfword or fullword containing the maximum output record length for the I/O operation. If OUTPUT (MEMORY) was specified, then this is taken to be a maximum length. In the case of OUTPUT(SERCOM,SPRINT,SPUNCH,WRITE, SUBROUTINE), MSG will break down the output buffer into smaller segments based upon the value of the expression at re. MSG will split lines at the last blank character in the buffer before the record length specified; if there is no blank, then MSG will split the line at the given length.

(4) FDUB=re

(Optional for the SUBROUTINE operator.) This operator supplies the location of an FDUB which will be passed to the subroutine.

PICTURE OPERATOR

The PICTURE operator can be used to convert numeric values to many different forms of character representation. The form of the output string is described by a "picture" specification similar to those of COBOL or PL/I.

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.

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, the decimal point does not appear if it is passed over by right zero suppression.

d,D Specifies literal insertion of a "." even if there is no following character.

, Specifies literal insertion of a comma; suppressed if not preceded and followed by a digit.

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)(9)[.(9)(Z)(P)]
```

where (x) indicate 0 or more occurrences of the "x" 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" 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" position left of the decimal. The pattern must provide at least one "Z" if the number is negative.

MSG EXAMPLES

The following examples illustrate some of the versatility of the MSG routine and macros:

```
PHRASE TAB(9),'<--'
PHRASE TAB(6),'|'
PHRASE TAB(1),'-->'
PHRASE END
```

generates:

```
"-->  |  <--"
```

A more comprehensive example in the form of a complete program is given on the following page. This program contains examples of the various uses of the OUTPUT operator.

```
MSGDEMO  CSECT
         REQU  TYPE=DEC
         ENTER R12,SA=SA1          Normal S-type entry sequence

*  Output with OUTPUT(SUBROUTINE(...))

         PMSG  OUTPUT(SUBROUTINE(ADDR=VSPUNCH)),' -> SPUNCH'

*  Output with WRITE(FDUB(...))

         LA    R1,MSOUNAME         We need a FDUB for this
         L     R15,=V(GETFD)       GETFD will give us one.
         BALR  R14,R15
         ST    R0,MSOUFDUB

         PMSG  OUTPUT(WRITE(FDUB=MSOUFDUB)),' -> WRITE(FDUB)'

*  Output with WRITE(MEMORY(...))

         PMSG  OUTPUT(MEMORY(ADDR=BUFFER,ORL=H30)),' -> MEMORY'
         SERCOM BUFFER,10          Print the buffer normally

*  Subroutine calling and return code checking

         PMSG  CALL(BR14),IF(*,EQ,0),' OK',ELSE,' Bad return code'

*  Miscellaneous examples

         LA    R2,147              Load register 2 with a number
         PMSG  ' Register 2 contains: ',DEC(R2,LEN=4)

         EXIT                      Normal S-type exit sequence

*  This is a very small subroutine

BR14     ENTER R12,SA=SA2,BASE=MSGDEMO
         EXIT  0,(0)

*  Data area follows:

SA1      DS    18A
SA2      DS    18A

H30      DC    H'30'
BUFFER   DS    CL30
MSOUNAME DC    C'*MSINK* '
MSOUFDUB DC    A(0)
VSPUNCH  DC    V(SPUNCH)

         END
```

The following program segment illustrates the use of MSG to format a quite complicated record:

```
            ...
            PMSG  OUTPUT(SPUNCH),STATMSG
            ...
     STATMSG  PHRASE (ONDATE),' ',(ONTIME)
            PHRASE ' ',WIDTH(6),DEC(CONNSECS)
            PHRASE ' ',IF(MSOURCE),'R',ELSE,'I'
            PHRASE ' ',WIDTH(4),(INODE,LEN=2)
            PHRASE ' ',(STYPE,LEN=2)
            PHRASE ' ',WIDTH(4),(RNODE,LEN=2)
            PHRASE ' ',WIDTH(12),(USERID)
            PHRASE ' ',BEGIN,FILL(C'0'),WIDTH(2),DEC(OPENRC),END
            PHRASE ' ',WIDTH(8),DEC(RECORDS_IN)
            PHRASE ' ',WIDTH(8),DEC(RECORDS_OUT)
            PHRASE ' ',WIDTH(10),DEC(BYTESI)
            PHRASE ' ',WIDTH(10),DEC(BYTESO)
            PHRASE ' ',WIDTH(4),(HOSTCODE,LEN=2)
            PHRASE ' ',(DEVJOB,LEN=4)
            PHRASE ' ',(OPEN_OPT,BLEN=OPEN_OPT_LEN)
            PHRASE END
```

May 1983

May 1983

STRUCTURED PROGRAMMING MACROS

The following section describes a set of macros which allow assembly language users to make use of structured programming techniques in coding their programs. Two types of control structures are provided by these macros: (1) decision structures which provide for selection among sections of code to follow, and (2) loop structures which provide a means of repeating sections of code.

In the first category are the IF-ELSEIF-ELSE-ENDIF and DOCASE-CASE-ELSECASE-ENDCASE macros. The IF...ENDIF macros permit a structure to be coded that selects among many parallel sections of code with many parallel conditions. The DOCASE...ENDCASE macros permit the selection of one of a number of groups of statements depending upon some integer value.

In the second category are the DO-ENDDO macros which provide several different looping structures. The DO-ENDDO macros provide an iteration structure that allows the loop termination condition to be specified at the beginning of the loop (on the DO macro) or at the end of the loop (on the ENDDO macro) or both. In addition, the FOR clause of the DO macro provides a labelless looping structure which is similar to the FORTRAN DO statement. The REDO, NEXTDO, and EXITDO macros provide ways of branching around in these looping structures without specifying labels.

In addition, the FLAGS, SET, and TEST macros allow the assembly language programmer to define, set, and test single bit program flags (switches) without remembering which flag belongs to which switch byte. The DEFCC macro allows the user to define additional conditions for comparisons in logical relations in IF and DO macros. Finally, the MACSET macro is provided to set various global options and parameters for the structured programming macros.

LOGICAL EXPRESSIONS

The IF and DO macros provide decision-making control structures and thus require specification of a condition for selection (IF) and repetition (DO). In a typical macro prototype like

```
[label]  IF    lexp
           .
           .
         ENDIF [label]
```

this condition is termed a logical expression lexp. A logical expression consists of a simple or compound condition which when evaluated has the value true or false.

Simple Conditions

Simple conditions are of two types: those which generate only a conditional branch instruction, and those which generate a compare instruction followed by a conditional branch instruction.

For the first type of simple condition which generates only a branch instruction, the user must have set the machine condition code appropriately with an instruction preceding the macro call. The logical expression lexp in this case has the form

```
cond
```

where cond specifies the condition to be tested and may be one of:

(a) EQ NE LT GT LE GE
(b) P M H L E O Z
(c) NP NM NH NL NE NO NZ
(d) POS ZERO NEG ONE ONES MIXED

where (a) are the FORTRAN compare operation mnemonics, (b) and (c) are the extended-branch mnemonic suffixes and their negations, and (d) are some spelled-out mnemonics. The DEFCC macro may be used to define additional condition mnemonics. An example lexp of this type is

```
LTR  R0,R15
IF   Z
```

The second type of simple condition specifies a logical valued expression of the form

```
operand1,cond,operand2,compare-op
```

which defines a relationship between two quantities. Two instructions are assembled, a compare instruction specified as

```
compare-op operand1,operand2
```

and a branch instruction to test the relation cond, defined above. Examples of simple conditions of this type are:

```
R1,LT,=H'4',CH
=C'ABC',EQ,STRING,CLC
0(R2),NE,X'FF',CLI
```

Normally, four operands are required to specify logical expressions of this form. However, for certain relations it is possible to omit one or more of these operands and appropriate default values will be used.

Two-Part Relations

Two-part relations specify two operands and are defined as follows:

```
register,cond
```

or

```
'instruction',cond
```

In the first form, an LTR instruction is assembled to test the specified register in order to set the condition code. The second form allows an arbitrary assembler instruction to be generated to set the condition code. All primes within the instruction must be doubled according to normal 360/370 assembler conventions. The second form is particularily useful for instructions that require more than two operands since the syntax for a lexp allows only two compare instruction operands (operand1 and operand2). Examples are:

```
IF    R2,NZ
IF    'CLM   R1,B''1000'',PREFIX',EQ
```

These are equivalent to

```
LTR   R2,R2
IF    NZ
```

and

```
CLM   R1,B'1000',PREFIX
IF    EQ
```

Three-Part Relations

Three-part relations specify three operands (compare-op is omitted). The compare instruction that is assembled is determined from the type of the operands according to the following table:

Relation	Compare-Op	Example
register,cond,register	CR	R1,EQ,R2
register,cond,fullword	C	R4,NE,=F'5'
register,cond,halfword	CH	R0,LT,=Y(256)
symbol,cond,self-defining term symbol,cond,'c' 'c',cond,symbol	CLI	0(R8),EQ,X'FF' BUFF,NE,'$' ' ',EQ,LINE
symbol,cond,symbol 'string',cond,symbol symbol,cond,'string'	CLC	X,NE,Z 'ESD',EQ,CARD+1 CMD(4),EQ,'STOP'
packed,cond,packed	CP	PSYM,NE,=P'-2'

In order for the macros to generate the appropriate compare instruction listed in the table above, the following requirements must be met:

(1) If an operand is defined by an EQU instruction (e.g., a register name), the definition must appear before it is used in a logical relation.
(2) If an operand is defined by a macro, its definition must appear before it is used in a logical relation.
(3) The operands symbol, fullword, halfword, and packed must have an assembler-type attribute other than "U" (unknown). This will be true if the operands are defined via DS or DC statements anywhere in the assembly.
(4) A self-defining term must have a value of less than or equal to 255 in order for a CLI instruction to be generated. An absolute symbol may be specified for operand2 if it is defined before being used in a logical relation.
(5) 'string' specifies a primed character string of more than one character. The macros will generate a literal for the primed string by prefixing the operand with "=C" in the CLC instruction.
(6) 'c' specifies a single character enclosed in primes.
(7) The operand packed is a symbol defined with the packed-decimal format.

If a three-part logical relation does not satisfy the above conditions (e.g., both operand1 and operand2 have unknown attributes), the compare instruction generated will be a fullword compare (C). This will most likely cause an assembly error. Hence, the compare instruction compare-op should be explicitly specified as part of the logical expression.

Flag Variables

If the logical expression specifies only one operand, then operand1 is assumed to specify a "flag variable" defined via the FLAGS macro, if it is not one of the predefined conditions. Flag variables are analogous to FORTRAN or PL/I logical variables. A flag variable is basically a symbol equated to a single bit in a byte of storage and has the value "1" (corresponding to the ON state) or "0" (corresponding to OFF). A test-under-mask (TM) instruction is generated for simple conditions of this type to test the state of the flag variable. The conditional branch instruction generated depends on the value of the flag variable to be tested. The ON state is tested simply by writing the flag variable name as the lexp, e.g.,

```
flag
```

The OFF state may be tested by preceding the variable flag with a not symbol (¬ or -), e.g.,

```
¬flag   (or -flag)
```

In this case, the condition is satisfied if the flag variable is 0 (OFF).

The syntax to specify the ON condition of a flag is identical to a simple condition of the first type, i.e., cond. Because of this, a flag variable must not have the same name as one of the predefined conditions. The FLAGS macro will print an error message if an attempt is made to redefine a condition as a flag variable. See the FLAGS macro for a description of how flag variables are defined and also how instructions to set and test them are formed.

Several flag variables may be tested with one TM instruction by specifying a flag expression flagexp of the form

```
flag1+flag2+...+flagn
```

or

```
flag1*flag2*...*flagn
```

The first form tests for the condition that at least one flag is ON; the second form tests for the condition that all of the flags are ON. The flags must be defined in the same byte by the FLAGS macro since the TM instruction tests only one byte of storage. The macros will enforce this restriction if the flags are defined before being used. For example, the macro

```
IF    BATCH*QUIT
```

will test for the condition that both the BATCH and QUIT flags are ON.

Individual bits of a byte may be tested in a logical expression (via the TM instruction), even if they are not defined by the FLAGS macro, by coding flag expressions flagexp of the form

```
bitn:scon
(bit1+bit2+...+bitn):scon
(bit1*bit2*...*bitn):scon
```

where "bitn" is a self-defining term or a symbol defined by an EQU pseudo-op and "scon" is any assembler base-displacement expression specifying the byte containing the bit(s) to test. The first form tests for a single bit being on, the second form tests for at least one bit being on, and the third form tests for all bits being on. Examples are:

```
IF    ¬GDCONCAT:GDSWS
IF    (GDEXBLN+GDEXELN):GDSWS2
IF    X'80':4(R1)
```

Program Interrupt Condition

The IF macro recognizes a special one-part logical expression that tests for a program interrupt in the preceding instruction. The syntax of the lexp is

```
IF    PGNT(type)
```

or

```
IF    ¬PGNT(type)        (or IF   -PGNT(type))
```

The IF macro generates a BPI macro using "type" as the first operand. If "type" is omitted, e.g., PGNT(), then PGNT(OPND) is defaulted. The PGNT() lexp is allowed only in simple IF statements. For example, the following sequence may be used to test for an OPND-type program interrupt:

```
L     R2,0(,R1)
IF    PGNT(OPND)
  .
  code to execute if there is a program interrupt
  in the LOAD instruction, e.g., if register 1
  contains an invalid address.
  .
ENDIF
```

See the description of the BPI macro in this volume for further details.

Compound Conditions

A compound condition is formed by combining simple conditions with the logical operators AND and OR. The form of a compound condition is thus:

```
(lexp1),lop,(lexp2),lop,(lexp3),...
```

where lop is either AND or OR. The simple conditions lexp1, lexp2, and lexp3 must be enclosed in parentheses because they consist of a variable number of operands (from one to four).

A compound condition consisting of simple conditions joined together by only AND operators is satisfied if and only if all simple conditions are satisfied. Failure of any simple condition results in a failure for the entire compound condition. Once a failure is detected, the remaining simple conditions are not evaluated.

A compound condition consisting of simple conditions joined together by only OR operators is satisfied if one (or more) simple conditions is satisfied. The entire compound condition fails if and only if all simple conditions are not satisfied. Once a success is detected, the remaining simple conditions are not tested.

Compound conditions may be formed by joining simple conditions together with both the AND and OR operators in one logical expression. The expression will be evaluated from left to right with the AND operator given higher precedence than OR. For example, the expression

```
(lexp1),OR,(lexp2),AND,(lexp3)
```

is satisfied if lexp1 is true or both lexp2 and lexp3 are true. It fails if lexp1 is false and either lexp2 or lexp3 or both are false. The normal order of evaluation of logical expressions may be changed by enclosing simple condition groups in parentheses. In the example above, the OR operator may be given higher precedence than the AND as follows:

```
((lexp1),OR,(lexp2)),AND,(lexp3)
```

This compound condition is satisfied if either lexp1 or lexp2, or both, is true and lexp3 is true.

Compound conditions may thus consist of compound conditions combined together with ANDs and ORs. This rule permits arbitrarily complex logical expressions to be formed. Some examples of compound conditions are:

```
(R1,NZ),AND,('ESD',EQ,CARD+1)
(R0,EQ,F2),OR,(R0,EQ,F5),OR,(R0,EQ,F7)
((R1,POS),AND,(R1,LE,F2)),OR,((R1,GE,F5),AND,(R1,LT,F9))
(BATCH),OR,(¬CMDMODE)
```

Note the compound condition formed by combining the two simple conditions consisting of the flag variables BATCH and CMDMODE.

The NOT logical operator may be used to negate a condition. In any context where lexp is valid, the form

```
NOT,(lexp)
```

may be used. The logical expression that is being negated must be enclosed in parentheses. For example,

```
NOT,(R1,Z)
(ATTN),AND,NOT,(BATCH)
(A,EQ,B),AND,(NOT,((C,EQ,D),OR,(E,EQ,F)))
```

Set Expressions

A simple type of "set" expression may be used in logical expression. The syntax is

```
value,setop,(set)
```

where

"value" is a register number or storage reference,
"setop" is IN, ¬IN, -IN, or NOTIN, and
"set" is a list of operands "S1,S2,...,Sn" that make up the set, where each "Sn" may be a range expression of the form (Si,...,Sj) meaning all elements between Si and Sj, inclusive.

The "set" expression is basically a shorthand method of writing a series of OR or AND compound expressions. Examples are:

```
R1,IN,(ONE,=H'3',(=F'7',...,=F'9'))
T,¬IN,(X,...,Y)
```

These are equivalent to

```
(R1,EQ,ONE),OR,(R1,EQ,=H'3'),OR,((R1,GE,=F'7'),AND,(R1,LE,=F'9'))
NOT,((T,GE,X),AND,(T,LE,Y))
```

IF, THEN, ELSE, ELSEIF, ENDIF

Macro Description

Purpose: To provide conditional if-then-else control structures for assembly language users. See also the DO-ENDDO and DOCASE-ENDCASE macro descriptions for iteration and selection control structures.

Prototype:

```
(1) [label]  IF    lexp
            [THEN]
               .
               user code to be executed if lexp
               is true
               .
             ENDIF [label]

(2) [label]  IF    lexp
            [THEN]
               .
               user code to be executed if lexp
               is true
               .
             ELSE
               .
               user code to be executed if lexp
               is false
               .
             ENDIF [label]

(3) [label]  IF    lexp1
            [THEN]
               .
               user code to be executed if lexp1
               is true
               .
             ELSEIF lexp2
            [THEN]
               .
               user code to be executed if lexp1
               is false and lexp2 is true
               .
             ELSEIF lexp3
            [THEN]
               .
               user code to be executed if lexp1
               and lexp2 are false and lexp3 is
               true
               .
               .
```

```
        as many ELSEIF statements as
        necessary
        .
      [ELSE
        .
        user code to be executed if none
        of the preceding lexps was true
        .]
       ENDIF [label]
```

Parameters:

lexp — specifies a logical expression which evaluates to either true or false. See the beginning of this section for a complete description of logical expressions.

These macros may change the condition code and the contents of the registers depending on the logical expressions specified.

Description: The simple IF of form (1) is used when the execution of one section of code depends on the value of one condition.

The IF...ELSE structure of form (2) is used when one of two sections of code is to be selected based on the value of one condition.

Form (3) illustrates an IF...ELSEIF structure which may be used when selecting among many parallel sections of code with many parallel conditions. The logical expressions (lexps) of the IF...ELSEIF structure are tested sequentially until the first successful test. The clause following the macro which specifies the successful condition is executed and then control proceeds with the first statement after the ENDIF macro. This construction could instead be coded equivalently with IF statements within the ELSE clause of the preceding IF. However, the ELSEIF construction permits the parallel program flow to be seen easily. Any number of ELSEIF clauses may be specified within the structure. The ELSE clause is optional and if present is executed if none of the preceding conditions were true.

The THEN macro is optional and may be used with any of the above forms to precede the true clause. However, its use is not recommended since it generates an extra statement label.

For all forms of the IF control structures, the optional label on the ENDIF must be the same as the label on the matching IF statement. If an outer level IF label is

specified, additional ENDIFs are automatically generated, but each with a MNOTE warning. If the label does not match any previous IF labels, only one ENDIF is produced and an error MNOTE is issued. The label on the ENDIF is optional even if a label is given on the preceding IF.

Examples: The first two examples are different ways of saying the same thing.

```
CLC   =C'YES',READAREA
IF    EQ
   CALL  SUB,(A,B,C)
ENDIF


IF    'YES',EQ,READAREA
   CALL  SUB,(A,B,C)
ENDIF


CALL  SUB,(D)
IF    R15,ZERO
   ST    R0,RESULT
ELSEIF (R15,EQ,=F'4'),OR,(R15,EQ,=F'8')
   CALL  WARNING
ELSE
   CALL  ERROR
ENDIF


IF    (R0,ZERO),AND,(R1,EQ,A),AND,(R2,EQ,B)
   ...
ELSEIF (A,GT,B,CLC),OR,((SORT),AND,(¬ORDERED))
   ...
ENDIF
```

The final example illustrates logical expressions consisting of compound conditions. Note the use of the flag variables SORT and ORDERED which must be defined by the FLAGS macro, described elsewhere in this section.

DOCASE, CASE, ELSECASE, ENDCASE

Macro Description

Purpose: To provide case selection control structures for assembly language users. See also the IF-ENDIF and DO-ENDDO macro descriptions for conditional and iteration control structures.

Prototype:

```
[label]     DOCASE caseno[,TREG=reg][,SCALE=num]
                   [,MAXCASE=max]
            CASE (icon,...)
               .
               statements to be executed if
               caseno equals any of the icons
               .
            CASE (icon,...)
               .
               statements to be executed if
               caseno equals any of the icons
               .
               .
               as many cases as required
               .
               .
           [ELSECASE
               .
               user code to be executed if caseno
               is less than zero or greater than
               the maximum icon, or caseno equal
               to some unspecified icon.
               .]
            ENDCASE
```

Parameters:

caseno is the number of the case to be selected. It may be an expression, the name of a halfword or fullword location, or a register number enclosed in parentheses.

icon specifies the case number of the following group of statements. It must be a nonnegative integer, or a predefined symbol equated to an appropriate integer value.

reg (optional) is a keyword parameter specifying the temporary register to be used in the code generated to select the proper case. GR0 may not be used. If omitted, GR14 is used unless caseno specifies a register, in which case the register containing caseno is

used. The contents of this register will be changed unless the scale factor num is 4. If num is not 1, 2, or 4, then reg must specify an even-odd register pair (GR14 and GR15 by default), since a divide instruction will be generated by the macro to compute the proper case. The MACSET macro, described elsewhere in this section, may be used to set the default temporary register to something other than GR14.

num (optional) is a keyword parameter specifying the scale factor used in numbering the cases. It must be a positive integer, or a predefined symbol equated to an appropriate integer value. This parameter is required only if the case number corresponding to num is not specified in a CASE macro.

max (optional) is a keyword parameter specifying the maximum case number to select. It must be a positive integer, or a predefined symbol equated to an appropriate integer value. This parameter is required only if the case number corresponding to max is not specified in a CASE macro.

Description: The CASE macros permit the selection of one of a number of groups of statements depending upon an integer value. Control passes to the appropriate case and then, unless otherwise exited, continues execution following the ENDCASE macro. If the control value is not specified in a CASE macro, execution proceeds with the ELSECASE clause. If there is no ELSECASE clause, execution continues past the ENDCASE.

The ELSECASE macro is optional. If present, the statements following are executed if caseno is less than zero or greater than the maximum case number, or equal to an unspecified case number. If ELSECASE is omitted, then no code is generated to check the caseno value for a valid case number, i.e., less than zero or greater than max. It is assumed that the user will test caseno before the DOCASE macro is executed.

Normally, cases are numbered consecutively beginning with zero (0,1,2,...) corresponding to a scale factor of one. However, cases may be numbered according to another scale factor; for example, SCALE=2 implies that cases are numbered as (0,2,4,...), SCALE=3 implies (0,3,6,...), and SCALE=4 implies (0,4,8,...). If the SCALE keyword parameter is not specified, the default scale factor num is taken as the smallest nonzero case number specified. If the case number corresponding to the desired scale factor

is not specified in a subsequent CASE macro, the parameter SCALE=num must be given on the DOCASE macro call.

Execution of the DOCASE macro begins with the prologue code generated by the ENDCASE macro. If there is an ELSECASE macro, the prologue code tests the control value caseno for a valid case number. The control is converted to a branch index and the proper case is selected by an indexed branch into the branch table generated by the prologue code. Because case selection is done with a branch table, the DOCASE structure should not be used if the case values are sparsely spread over a large region. The IF statement with appropriate ELSEIF clauses should be used instead.

Example: The following example illustrates the CASE structure.

```
CALL  GETLST,(UNIT,LASTLNR)
DOCASE (R15)              Test return code
CASE  (0)                 RC=0
   ...OK...
CASE  (4)                 RC=4
   ...File empty...
CASE  (12,24)             RC=12 or 24
   ...File doesn't exist or no access...
ELSECASE                  RC=8,16, or 20
   ...Deadlock, etc...
   (case < 0 or = 8,16,20 or > 24)
ENDCASE
```

Note that the CASE macro computes SCALE=4 and MAXCASE=24. Case numbers 8, 16, and 20 are missing and will cause a branch to the ELSECASE clause if selected.

DO, ENDDO

Macro Description

Purpose: To provide do-end iteration control structures for assembly language users. See also the IF-ENDIF and DOCASE-ENDCASE macro descriptions for conditional and selection control structures.

Prototype:

```
[label]      DO [iterargs][loopargs][SIGNAL=signal]
               .
             user code
               .
             ENDDO [loopargs]
```

Parameters:

loopargs specifies the conditions for repetition and may be given on either the DO or ENDDO macros or both. If omitted from the ENDDO macro, an unconditional branch back to the front of the loop is generated. Otherwise, the type of loop termination test generated is as follows for the following cases of loopargs:

COUNT=reg	generates a BCT instruction using register reg.
BXLE=(r1,r3)	generates a BXLE instruction using the registers specified.
BXH=(r1,r3)	generates a BXH instruction using the registers specified.
WHILE=(lexp)	generates the appropriate code to evaluate the logical expression lexp.
UNTIL=(lexp)	generates the appropriate code to evaluate the logical expression lexp.

lexp specifies a logical valued expression which evaluates to either true or false. Because lexp may consist of more than one operand, it must be enclosed in parentheses to conform to the keyword syntax of macro parameters. However, if lexp is either a predefined condition cond or a user defined flag variable flag, then it must not be enclosed in parentheses, since both cond and flag are specified as single operands. For example,

```
WHILE=cond
UNTIL=flag
```

See the beginning of this section for a complete description of lexp, cond, and flag.

iterargs specifies an iteration control statement of the form:

```
FOR=reg[,FROM=beg],TO=end[,BY=incr]
```

which implements the equivalent of a FORTRAN DO loop with the following parameters:

reg specifies a register number to be used as the loop control index. The contents of this register will change.

beg (optional) is a keyword parameter specifying the beginning value for the loop control index reg. It may be a self-defining term, an expression, the name of a location, or a register number enclosed within parentheses. If beg is omitted, then the initial value is assumed to be in reg.

end is a keyword parameter specifying the test value for the termination of the DO FOR loop. It may be a self-defining term, an expression, the name of a location, or a register number enclosed within parentheses.

incr (optional) is a keyword parameter specifying the increment to be added to the loop control index reg at the end of each iteration. It may be a self-defining term, an expression, the name of a location, or a register number enclosed within parentheses. If incr is omitted, then a default increment of 1 will be used. An explicit sign may precede incr (e.g., BY=-1) to subtract the increment (decrement) at each iteration. In this case, end should be less than (or equal) to beg. The minus sign must be specified in order to decrement the loop; specifying a location containing a negative value is not valid.

signal specifies a flag variable defined via the FLAGS macro or a character string enclosed in primes (e.g., 'BIG LOOP') which identifies the loop for subsequent embedded REDO, NEXTDO, or EXITDO macro statements. A list of signals may be specified by enclosing them (separated by commas) in parentheses.

Description: The DO macro provides an iteration control structure with the termination condition specified at either the beginning of the loop (on the DO macro) or at the end of the loop (on the ENDDO macro) or both. It is possible to specify more than one loop termination condition on a single macro call (DO or ENDDO) although due to the keyword syntax, the loopargs left-hand sides must be different (e.g., DO COUNT=R1,COUNT=R2 is not valid). If more than one looparg is specified, all conditions must be satisfied before the iteration will continue. The keyword conditions will be processed in the order listed in the loopargs description (i.e., COUNT, BXLE, BXH, WHILE, UNTIL in that order).

If the arguments are omitted from both the DO and ENDDO macros, an unconditional branch to the beginning of the loop is generated at the ENDDO macro, in effect, an endless loop.

The WHILE and UNTIL clauses provide similar structures except that the condition for repetition is expressed in opposite manners, e.g., WHILE=EQ is the same as UNTIL=NE. The loop iteration will continue when the WHILE condition is true and terminates when the UNTIL condition is true. Both the WHILE and UNTIL clauses may be specified on either the DO or ENDDO macros or both.

The COUNT, BXLE, and BXH parameters specify the type of branch instruction (BCT, BXLE, and BXH, respectively) to be generated to perform the iteration. If specified on the ENDDO macro, a branch to the DO macro is generated. However, if specified on the DO macro, two branch instructions are generated. The first is a conditional branch using the specified opcode and register(s) around the next instruction (into the body of the loop), followed by an unconditional branch to the statement following the ENDDO macro (out of the loop).

The FOR clause of the DO macro (iterargs specified) provides an iteration control structure similar to the FORTRAN DO loop. The register reg is initialized to the value beg and compared to the terminating value end. If reg is less than or equal to end when incr is greater than zero, or if reg is greater than or equal to end when incr is less than zero, then the statements in the range of the loop are executed. At the end of each iteration, reg is incremented by incr and a branch is made to the test at the beginning of the loop. Note that, unlike FORTRAN, the body of the loop may not be executed if the test condition is satisfied initially. It is also possible to specify loopargs with iterargs (the FOR clause) on the DO macro, but the test specified by

loopargs will be performed after the iterargs clause is processed.

Branching within a loop may be performed with the REDO, EXITDO, and NEXTDO macros, described elsewhere in this writeup.

The SIGNAL clause on the DO macro serves two functions. First, it provides a means of identifying the loop to be used by REDO, NEXTDO, or EXITDO macro statements which may be nested within several embedded loops. Second, it provides a means of recording a special condition which caused the loop termination. Either form of signal (flag variable or character string) may be used to identify the loop for REDO, NEXTDO, or EXITDO statements. However, the use of a flag variable as the signal specification allows the specific condition that caused the loop termination to be recorded. Each flag variable specified in the SIGNAL clause is set to the false (OFF) state at the beginning of the loop. The execution of an EXITDO macro statement specifying one of these signal flags will cause the flag to be set to true (ON) and the loop to be exited. The REDO and NEXTDO macro statements do not alter the value of any of these flag variables.

Examples: The following examples illustrate several of the available DO loop structures.

```
     L     R10,LISTHEAD
     DO ,  search for the end of a linked list
       LR    R9,R10          Save predecessor
       L     R10,NXTPTR      Follow link
       LTR   R10,R10         Continue until it's null.
     ENDDO WHILE=NZ          (or ENDDO UNTIL=ZERO)
```

Note that in the above example the test for the loop termination (when the link pointer is null) is made at the end of the loop. Below is a similar example but with the test made at the beginning of the loop.

```
     L     R10,LISTHEAD
     DO WHILE=(R10,NZ)
       LR    R9,R10     Save predecessor
       L     R10,NXTPTR Follow link
     ENDDO              Continue while link is nonzero
```

In the above example, the statements in the body of the loop will not be executed if the condition is satisfied initially, unlike the previous example.

```
LH    R2,LEN
LA    R3,AREA-1
DO ,  look for last nonblank character in AREA
  LA    R1,0(R2,R3)
  CLI   0(R1),C' '
  EXITDO NE
ENDDO COUNT=R2
```

The following is a similar example with termination conditions specified on both the DO and ENDDO macros.

```
LH    R2,LEN
LA    R1,AREA-1(R2)      Address of last character
DO WHILE=(0(R1),EQ,' ')
  S     R1,=F'1'         Back up to next character
ENDDO COUNT=R2
```

The following example illustrates the FOR clause of the DO macro. The loop will be performed twelve times.

```
DO FOR=R8,FROM=1,TO=12
  ST    R8,LINCT
  CALL  BLOKLETR,(CHARS,LINCT,OUTPUT,FLEN)
  SPRINT OUTA,OLEN
ENDDO
```

The following example illustrates a routine to process commands coded using several of the structured programming macros.

```
CALL  READCMD,(COMMAND)
DO WHILE=((R15,ZERO),                                  +
      AND,('STOP ',NE,COMMAND),                        +
      AND,('RETURN ',NE,COMMAND))
  LA    R1,CMDTAB
  DO FOR=R2,FROM=1,TO=NUMCMDS
    IF   COMMAND,EQ,0(R1)
      ...process command...
      EXITDO
    ENDIF
    LA    R1,CMDTABL(,R1)
  ENDDO
  CALL  READCMD,(COMMAND)
ENDDO
CALL  SYSTEM
```

The following example illustrates the SIGNAL clause of a DO macro.

```
LA    R1,TABLE
DO WHILE=(R1,LT,ATABEND),SIGNAL=FOUND
  IF ITEM,EQ,0(R1)
    EXITDO SIGNAL=FOUND
  ENDIF
  LA    R1,L'ITEM(,R1)
ENDDO
IF ¬FOUND
  PMSG ' ***Item "',(ITEM),'" not found***'
  EXIT 4
ENDIF
```

REDO, EXITDO, NEXTDO

Macro Description

Purpose: To provide labelless branching within do-end iteration control structures for assembly language users. See the DO-ENDDO macro descriptions for the available iteration control structures.

Prototype:

```
REDO    [lexp][SIGNAL=signal]
EXITDO  [lexp][SIGNAL=signal]
NEXTDO  [lexp][SIGNAL=signal]
```

Parameters:

- lexp — specifies a logical expression which evaluates to either true or false. See the beginning of this section for a complete description of logical expressions.
- signal — specifies a flag variable defined via the FLAGS macro or a character string enclosed in primes (e.g., 'BIG LOOP') which identifies the loop to be repeated (REDO or NEXTDO) or to be exited (EXITDO). The signal must be specified in the SIGNAL clause of an enclosing DO loop.

These macros may change the condition code and the contents of the registers depending on the logical expressions specified.

Description: The REDO macro causes execution to continue with the statement immediately following the innermost enclosing DO macro if lexp is true. The loop index incrementing for the FOR clause of the DO macro and the loop termination testing on the DO macro are skipped. A branch to the statement following the DO macro is generated.

The EXITDO macro causes execution to continue with the statement immediately following the end of the innermost enclosing DO-ENDDO macros if lexp is true. A branch to the statement following the ENDDO macro is generated.

The NEXTDO macro causes execution to continue with the loop terminating statement (i.e., ENDDO) of the innermost enclosing DO macro if lexp is true. A branch to the ENDDO macro is generated. This macro will cause the next loop iteration to take place if the loop condition is successfully met.

If lexp is omitted from the REDO, EXITDO, or NEXTDO macro statement, an unconditional branch will be generated. Branching is normally done at the current nesting level.

The SIGNAL clause may be specified to identify which of several enclosing loops is to be repeated or exited. If no signal is specified, the branch is made at the current nesting level. If the signal is a character string, the enclosing loop which specifies the signal in its SIGNAL clause will be repeated (if REDO or NEXTDO) or exited (if EXITDO). If the signal is a flag variable defined via the FLAGS macro, the EXITDO macro will set the flag to true (ON) and the enclosing loop which specified the flag in its SIGNAL clause will be exited. The REDO and NEXTDO macro statements will not change the value of the signal flag; it serves only to identify the loop.

Example:

```
LH    R2,LEN
LA    R3,AREA-1
DO ,  look for last nonblank character in AREA
  LA    R1,0(R2,R3)
  CLI   0(R1),C' '
  EXITDO NE
ENDDO COUNT=R2
```

The following example illustrates the use of the SIGNAL clause for identifying the loop to be exited.

```
DO FOR=R1,FROM=1,TO=N,SIGNAL='BIG LOOP'
  DO FOR=R2,FROM=1,TO=M
     ...
    EXITDO SIGNAL='BIG LOOP'
     ...
  ENDDO
ENDDO
```

DEFCC

Macro Description

Purpose: To define additional conditions for comparisons in logical relations used by the structured-programming macros.

Prototype: DEFCC cond,value[,TYPE=compare-op]

Parameters:

- cond — specifies the name of the condition mnemonic being defined.
- value — is either a constant in the range 0-15 or the name of a predefined condition which specifies the numeric value of the condition.
- compare-op — (optional) specifies an assembler instruction opcode to be used in generating the comparison for three-part logical relations.

Description: The DEFCC macro allows the user to define condition mnemonics for comparisons in addition to the predefined conditions (EQ, NE, etc.). The optional parameter TYPE may be specified to associate a particular assembler instruction opcode compare-op with the condition being defined. In this case, for three-part logical relations of the form

```
operand1,cond,operand2
```

the compare instruction generated would be the specified compare-op. If no instruction is specified, the compare opcode is determined from context, if possible; otherwise, it defaults to C. See the beginning of this section for a description of the predefined conditions and logical relations.

Example: The following examples illustrate several DEFCC macro calls and some typical logical relations using the user-defined conditions.

```
DEFCC EQUAL,EQ
DEFCC LEQ,8,TYPE=CL
DEFCC ON,O,TYPE=TM
  ...
IF R1,EQUAL,=F'1'
  ...
IF R2,LEQ,=CL4'WMTS'
```

```
  ...
IF GDSWS,ON,X'02'
```

The above IF statements are equivalent to the following:

```
C   R1,=F'1'
IF  EQ
  ...
CL  R2,=CL4'WMTS'
IF  EQ
  ...
TM  GDSWS,X'02'
IF  O
```

FLAGS

Macro Description

Purpose: To define one or more symbols as single bit program flag variables to be used with the structured programming macros. See also the SET and TEST macro descriptions.

Prototype:
```
[label]     FLAGS flag1,flag2,...,flagn
                  [,DS=NO][,DC=YES]
```

Parameters:

flagi — specifies a symbol to be defined as a program switch. An initial value may be assigned to the flag variable by enclosing the symbol flagi in parentheses as follows:

```
(flagi,b)
```

where b is ON or OFF. If b is omitted, then OFF is used. If flagi is "*", then the next available flag bit position in label will be reserved but unnamed. There may be at most 248 symbols defined as flags in one macro call.

DS=NO — (optional) specifies that a DS (define storage) pseudo-op is not to be assembled for label to reserve the appropriate amount of storage for the specified flag variables. In this case the user must code a separate instruction to define the symbol label. If DS is omitted, then storage is reserved unless a MACSET macro has specified otherwise.

DC=YES — (optional) specifies that a DC (define constant) pseudo-op is to be assembled for label to initialize the specified flag variables. Each flag variable will have a zero initial value unless specified otherwise. If DC is omitted, no storage is initialized unless initial values are given for one or more flag variables, in which case all flags will be initialized.

Description: The FLAGS macro allows the assembly language user to define program flag variables to be used in structured programming macros so it is not necessary to remember which flag belongs to which switch byte. The name field of the macro call, label, designates the location to

contain the flag variables to be defined. Each symbol flagi defined will be assigned to one bit beginning with the high-order bit of the first byte of label. One byte of storage is required for every eight flag variables defined. Flag variables are defined by equating each symbol flagi to a byte beginning with location label, specifying the assigned bit position as the length attribute of the symbol flagi. For example, the code assembled to define n symbols as flag variables is:

```
flag1   EQU   label,X'80'          bit 0
flag2   EQU   label,X'40'          bit 1
        ...
flag8   EQU   label,X'01'          bit 7
flag9   EQU   label+1,X'80'        bit 8
        ...
flagn   EQU   label+(n-1)/8,X'zz'  bit n-1
```

where zz is the bit mask for flagn. This definition allows a flag to be set with the instruction

```
OI    flag,L'flag
```

or reset by

```
NI    flag,255-L'flag
```

and tested via

```
TM    flag,L'flag
```

The SET and TEST macros provide convenient ways of generating these instructions. See their macro descriptions for details; see also the introduction to this section for details on the use of flag variables with the structured programming macros.

The label field may be omitted from a FLAGS macro call. In this case, the flag variables will be assigned beginning with the next available bit position in the location specified by the last nonblank label field of a previous FLAGS macro. Additional storage will be reserved as required. Because of this, user code should not intervene between such FLAGS macro calls.

Example: The following example shows both a sample macro call and the corresponding instructions assembled. Note that only the flag variable PROMPT is initialized to the ON state (the bit corresponding to the mask X'20' is 1), the remaining three bits are initialized to the OFF state. The low-order four bits of the byte SWS are unused.

```
SWS    FLAGS BATCH,          On if batch mode     +
             ATTN,           On if attention      +
             (PROMPT,ON),    On to prompt user    +
             TERSE           On for short output
```

generates

```
SWS    DC    BL1'00100000'
BATCH  EQU   SWS,X'80'
ATTN   EQU   SWS,X'40'
PROMPT EQU   SWS,X'20'
TERSE  EQU   SWS,X'10'
```

Below is an equivalent example which defines the same flags in separate macro calls instead of using one macro call with several continued statements.

```
SWS  FLAGS  BATCH
     FLAGS  ATTN
     FLAGS  (PROMPT,ON)
     FLAGS  TERSE
```

FLAGVAL

Macro Description

Purpose: To initialize one or more bytes to specified flag mask values.

Prototype: label FLAGVAL flagexp1,flagexp2,...,flagexpn

Parameters:

flagval specifies any valid flag expression. Each operand allocates and initializes one byte of storage. If flagexp is omitted or is "*", a zero-valued byte is allocated. If flagexp specifies a self-defining term, a byte with that value will be allocated.

Examples:

```
WAITMASK FLAGVAL WAITING
         ...
         NC      0(1,R1),WAITMASK

DEFFLAGS FLAGVAL TERM+PROMPT,0
         ...
         MVC     FLAGS(2),DEFFLAGS
```

SET

Macro Description

Purpose: To set a flag variable defined via the FLAGS macro ON or OFF. See also the FLAGS and TEST macro descriptions.

Prototype: [label] SET flag-list[,{ON|OFF}]

Parameters:

flag-list specifies a list of flag variables (separated by commas) to be set. Each flag must be defined using the FLAGS macro.

Description: The SET macro is used to change the state of one or more flag variables as defined via the FLAGS macro. If the last positional parameter is ON or not OFF, the following instruction is assembled to set each flag variable flag to 1:

```
label   OI    flag,L'flag
```

If the last parameter to the macro is OFF, then an instruction of the following form is assembled to set each flag variable flag to 0:

```
label   NI    flag,255-L'flag
```

A separate instruction will be generated to set each flag specified. If all flag variables are defined via FLAGS macros before the SET macro invocation, the macro will detect which flags are defined in the same bytes and generate as few instructions as necessary to set them.

See the introduction to this section for further details on flag variables.

Examples: The following two examples are equivalent.

```
          SET   ATTN,ON

          SET   ATTN

          CALL  CANREPLY
          IF    R15,NZ
            SET   BATCH,QUIT   User in batch mode
          ENDIF
            .
            .
```

```
          IF    BATCH          Batch user?
            CALL  QUIT         Yes, $SIGNOFF
          ELSE ,         Allow $RESTART from terminal
            CALL  ERROR
          ENDIF
            .
            .

     SWS  FLAGS (BATCH,OFF)    On if batch user
          FLAGS (QUIT,OFF)
```

This example illustrates the use of the flag variable BATCH in SET, IF, and FLAGS macro calls.

TEST

Macro Description

Purpose: To test if a flag variable defined via the FLAGS macro is ON or OFF. See also the FLAGS and SET macro descriptions.

Prototype: [label] TEST flag[,ON=tloc][,OFF=floc][,MIXED=mloc]

Parameters:

flag	specifies the flag expression to be tested.
tloc	(optional) is the location to branch to if the flag variable is ON.
floc	(optional) is the location to branch to if the flag variable is OFF.
mloc	(optional) is the location to branch to if the flag bits are not all ON or OFF.

Description: The TEST macro generates a test-under-mask (TM) instruction to check the state of the specified flag variable. The instruction assembled has the form:

```
label   TM    flag,L'flag
```

where flag is a flag variable defined via the FLAGS macro. If the keyword ON is specified, a branch on condition one (BO) instruction to the specified location tloc is assembled, corresponding to the flag variable state ON. If the keyword OFF is specified, a branch on condition zero (BZ) instruction to the specified location floc is assembled, corresponding to the flag variable state OFF. If the keyword MIXED is specified, a branch-on-mixed (BM) instruction to the specified location mloc is assembled. In this case, flag should be a flag expression specifying more than one flag bit to be tested. All three keywords may be specified on one macro call.

Note that flag variables may also be tested by specifying them as conditions in logical expressions (lexps) of IF and DO macros. See the introduction to this section for further details on flag variables.

Examples: The following three examples are equivalent.

```
TM    ATTN,L'ATTN
BO    ATTNXIT
```

```
TEST  ATTN
BO    ATTNXIT

TEST  ATTN,ON=ATTNXIT
```

May 1983

MACSET

Macro Description

Purpose: To establish global switch settings and parameter values to be used in following macros.

Prototype: MACSET keyword[,keyword,...]

Parameters:

keyword specifies a keyword option. See below for the description of the available keywords.

Keywords: LITADDR={YES|NO}

This keyword sets the global SETB (GBLB) switch &LITADDR to 1 if the keyword value is YES, or to 0 if the right-hand side is NO. This switch is tested by certain macros, such as CALL, to see whether it can generate a literal instead of an inline adcon. See also the LITADDR macro description.

LABTYPE={STMT|LINE}[(NR)]

This keyword sets the global SETC (GBLC) symbol &LBLTYPE to the value of the keyword right-hand side if it is STMT or LINE. This symbol is tested by the conditional and iteration control structure macros (IF, DOCASE, and DO) when generating internal labels for these control structures. If the symbol is STMT (the default), the labels generated for a control structure will have as a suffix the source statement number of the macro beginning the control group. For example, a DO macro at statement number 123 will generate an internal label of the form "#DO123". If the symbol is LINE, then labels generated for a control structure will have as a suffix the source line number of the macro beginning the control group. For example, a DO macro at source file line number 123.000 will generate an internal label of the form "DO#123". Note that the label prefix is different for the two cases to distinguish between statement and line numbers. Fractional line numbers are encoded by replacing the decimal point with a pound sign. For example, a DO macro at line number 88.52 will generate an internal label of the form "DO#88#52".

If (NR) is appended to STMT or LINE, then all control-structure internal labels except for the IFxxx and DOxxx types will be generated as nonrelocatable symbols with no SYM record entries produced for them.

LABPFX={XXc|cXX}

This keyword may be used to change the form of the label prefix for internal labels generated by the macros. "c" may be any one of "#", "@", "_", or "$". For example, the macro

```
MACSET LABPFX=#XX,LABTYPE=LINE
```

generates labels of the form "#IFlinenumber" instead of the default "IF#linenumber". This keyword may be used to isolate all internal labels into one section of the cross-reference listing (if the postprocessor *PEXIT is not used to remove such labels).

CASEREG=reg

This keyword sets the default temporary register to be used in computing the case index for the DOCASE macro to reg. By default, GR14 is used. See the DOCASE macro description for further details.

FLAGDS={YES|NO}

This keyword specifies whether the FLAGS macro is to generate a DS (define storage) instruction to reserve storage for the specified flag variables or not. By default, a DS instruction will be assembled for each FLAGS macro. See the FLAGS macro description for further details.

EXLIT={YES|NO}

If the EXLIT=YES option is specified, the syntax of the execute (EX) instruction is extended to allow specification of the executed instruction as a "literal" operand. For example,

```
          EX  R1,=I'MVC PAR(0),0(R2)'
```

may be coded in addition to

```
          EX  R1,MVC1
            ...
MVC1      MVC PAR(0),0(R2)
```

The executed "literal" instructions will be included at the end of the literal pool that is generated by the next LTORG or END statement. This literal pool must be in the same control section as the EX instructions which refer to it. Since the assembler does not support literal instructions, the EXLIT option causes the EX instruction to be OPSYNed to a macro which processes the "literal" instruction. The EXLIT=NO option may be specified to restore the original definition of the execute instruction.

If the EXLIT=YES option is used allow "literal" instructions on EX instructions, then all ORG statements in the assembly must specify nonblank operands. If the operand is omitted from the ORG statement, then the assembler will be unable to resolve the reference; hence the error message

```
IEV080 ***ERROR*** STATEMENT IS UNRESOLVABLE
```

will be printed for the forward referencing ORG generated by the extended EX instructions.

Example:

```
MACSET LITADDR=YES,LABTYPE=LINE,EXLIT=YES
```

May 1983

May 1983

IOH

INTRODUCTION

IOH is an input/output conversion package that provides format-directed input and output for 360/370-assembler language programs and programs using the Plot Description System. Programs written in FORTRAN must use a different format-directed input/output package, which is similar in appearance to IOH, but different in detail. This package is described in the section "FORTRAN I/O Library" in MTS Volume 6, FORTRAN in MTS.

A Simple Case

To illustrate the input and output of data items under format conversion, consider a small portion of an assembly language program that reads an integer N, computes its factorial N!, and prints the result:

```
            RDFMT  FMT1,(N,0)            Read N
            L      GR2,N
            SR     GR3,GR3
            LA     GR4,1
LOOP        MR     GR3,GR2               Compute N!
            BCT    GR2,LOOP
            ST     GR4,RESULT
            PRFMT  FMT1,(RESULT,0)       Print result
             .
             .
N           DS     F
RESULT      DS     F
FMT1        DC     C'I10*'
```

To perform format-directed input and output in this program, three things must be specified:

(1) the logical I/O units to be used,
(2) the list of data items that are to be read or written, and
(3) the formats in which the data items are to appear.

Since the calling sequences for IOH are rather complex, there is a set of macros in the system macro library (*SYSMAC) to generate these calling sequences. These macros are easy to use and since it is uncommon to write the calling sequences by hand, all examples given will

use the macros. The complete calling sequences are given in Appendix B to this section.

The most commonly used macros are:

RDFMT - generates a call to SCARDS
PRFMT - generates a call to SPRINT
PCFMT - generates a call to SPUNCH

The sample program will read the value N from SCARDS and print the result on SPRINT. Thus, to read N through logical I/O unit SCARDS, the macro RDFMT is used:

```
RDFMT  FMT,(N,0)
```

The first parameter FMT specifies the location of the format, and the second parameter (N,0) is the list of data items to be read. In this case, there is one item in the list, N. Since the list can be of any length, the list must have a terminator; this terminator is a zero item in the list.

The format is supplied at the location FMT. The format is a character string that specifies how the data items are to be treated. The type of conversion desired and the column range in which the data items will appear must be specified. The type of conversion is specified by giving a format control character. In the above example, the format control character is "I" which specifies that the data items are to be treated as a decimal integers. The number of columns that a data item occupies in the input line is called the external field width. The external field width follows the control character. In the above example, N will occupy 10 columns. A format always starts at column 1, hence a format beginning with I10 indicates columns 1 to 10.

The format so far is "I10". One more thing remains. A control character must be added that will terminate the format character string. This control character is called the format terminator and is the asterisk "*". The final format is

```
I10*
```

Now that the format has been specified, it must be inserted into the program. Since a format is a string of characters, it can be inserted as a character constant:

```
FMT    DC   C'I10*'
```

The control character I determines which characters are legal in the input field. Since I specifies an integer, the only legal characters are the digits 0 to 9, the + and - signs, and blanks. Decimal points, equal signs, etc., are not legal; if they are found, an error comment will be given and the conversion will be terminated. The data item may appear anywhere in the input field.

When the data item is read from the input line, it is converted from its external character form into its corresponding internal form. For an integer, the internal form is a binary integer. The size of the internal form is called the internal field width. For the above example, the internal field width is a fullword (4 bytes).

Looking at the program again, once the number N has been read and its factorial computed, it is necessary to print the result. The statement to do this is:

```
PRFMT   FMT,(RESULT,0)
```

The only differences between this statement and the RDFMT statement are the name of the macro used and the location named in the list, in this case RESULT. The action, however, is different. The contents of RESULT are printed on SPRINT. The same format is used, but the meaning is slightly different. Here I10 means convert the number from a binary integer into its character representation, and place it in the next 10 columns of the print line, right-justified, and fill the unused columns to the left with blanks. The "*" not only terminates the format, but also causes the line to be printed. The printed line is blanked out except for the data items inserted.

Format Terms

Formats are used to specify how data elements are to be converted between the internal and external forms. For input, the line image is the input line containing the data items to be read; these data items are taken from the line image area and converted from their external character form into the appropriate internal form by IOH. For output, the line image is the output line containing the data items to be printed; these data items are converted from their internal forms to their appropriate external character form and placed in the line image area by IOH.

In the preceding example, the format "I10*" was used to read one integer. Suppose that 3 integers were to be read; the first is in the first 10 columns of the line image, the second in the next 5 columns, and the last in the next 20 columns. A possible statement to do this would be:

```
RDFMT  FMT1,(A,B,C,0)
```

and the format would be:

```
FMT1   DC   C'I10,I5,I20*'
```

Each subset of characters, separated by commas and by the final asterisk, is called a format term. Items in the list and terms in the format are matched. The first format term corresponds to the first item

in the list, A. The integer read will be placed in the location named A. Similarly, the integer specified by I5 is placed into B, and the integer specified by I20 is placed into C.

TYPES OF FORMAT SPECIFICATIONS

The following paragraphs describe the most common types of format specifications available. These paragraphs are only introductory in nature. For the complete descriptions of each of the format specifications, see Appendix A to this section.

Integer

The control character for integer conversion is "I". The format term is given in the form

```
Iw
```

where "w" is the external field width. The default internal field width is a fullword. To specify a halfword integer, a H modifier can be used in the format term. The H modifier specifies that a halfword should be used as the internal field width. For example,

```
HI8
```

For input, the next "w" columns of the input line image are scanned for the number. Blanks are ignored; they are not considered the same as zeros. Thus, the character "3" placed anywhere in the next "w" columns will cause the value 3 to be read. If "w" is omitted on input, standard format input is assumed (see the subsection "Standard Format I/O"); in this case, the entire input line image is scanned for a legal integer terminated by a blank, comma, or right parenthesis.

On output, the number is converted to an integer and placed right-justified in the next "w" columns of the output record. If "w" is omitted, the external field width defaults to 8.

Floating-Point (F-type)

There are two ways to specify floating-point numbers in a format: F-type and E-type. F-type specifies integers and fractions. Examples of numbers of this type are 30.12, 3.14159, .025, and 10000. The format term used to read or print F-type floating-point numbers is:

```
Fm.n.w
```

where "F" is the control character, "w" is the total external field width, "m" is the number of digits before the decimal point, and "n" is the number of digits after the decimal point. The period is used as a delimiter for the three field widths and must be present.

For output, the "F" format causes the number to be converted, rounded to "n" digits after the decimal point, and placed right-justified in the next "w" columns of the output line image. If "w" is omitted, it defaults to m+n+2. For example:

```
F5.3.10
```

is the same as

```
F5.3
```

Printing floating-point numbers according to this format would give such output as:

```
3.150     -1235.000     .000     10.520
```

If no digits are desired after the decimal point, the format could be written as "F10.0". It is possible to suppress the decimal point using the M modifier in the format term. For example,

```
MF10.0
```

When an F format is used to read a number, the action is slightly different. If the number has a decimal point (e.g., 30.12), the "n" has no effect, and the number is taken as read. If there is no decimal point, "n" specifies where the decimal point belongs. The decimal point is assumed to be "n" digits to the left of the rightmost digit. For example, according to the format F5.3, the following three numbers are the same:

```
3.142   3142   3 14 2
```

Note that blanks are ignored. Also the following are the same:

```
.003   3
```

By default, floating-point numbers are stored internally in short-precision (4-byte) form. To specify long-precision (8-byte) form, the D modifier must be included in the format term. For example,

```
DF5.3
```

The following points should be noted:

(1) It is the responsibility of the user to make sure that when a floating-point conversion is specified, the associated list

element actually contains a floating-point number. Similarly, if integer conversion is specified, the number to printed should be an integer. Since there is no possible way to determine whether the contents of a storage location is an integer, a floating-point number, or any other item, IOH has no choice but to convert the number according to the specification given. For example, the fullword containing X'41100000' can be interpreted as meaning the floating-point number 1.0 or the integer 1091567616.

(2) The user must be sure to specify a field width that will be large enough for the number when converted. For example, the specification F6.3 allows 6 columns for the integer part of the number. This means that the maximum number that can be printed by this format is 999999.999. If larger numbers are used, an error will occur and the program will be terminated. This action can be changed by setting a switch; see the subsection "Additional Entry Points to IOH."

(3) The user must be sure to specify the correct internal size of the number. If the variable is long-precision floating-point, the "D" modifier must be specified for input or output.

Floating-Point (E-type)

The second way to specify floating-point numbers is by using a fraction and an exponent. Examples of numbers of this type are

```
30.12E 02   .314159E+01   .025E-01   1.E+05
```

The first means 30.12 times 10 to the power 2, the second means .314159 times 10 to the first power, etc. The format term used to read or write E-type floating-point numbers is

```
Em.n.w
```

where each of the fields has the same meaning as for the F-type specification:

m - number of digits before the decimal point
n - number of digits after the decimal point
w - external field width

Because of the form of the number, the external field width specification on output must be large enough to include not only the number of digits before and after the decimal point, but also the decimal point, the sign of the number, the "E", the sign of the exponent, and the two digits of the exponent. If "w" is not given, it defaults to the value m+n+6.

For input, the entire number, both fraction and exponent, must be in the field specified. If there is no decimal point in the input field, an assumed decimal point is placed "n" places to the left of the rightmost digit of the fractional part. Thus, according to format E6.3

```
39764E2
```

is the same as

```
39.764E2
```

Character

The control character for reading or printing characters is "C". The format term is given in the form

```
Cw.n
```

where "w" is the external field width and "n" is the internal field width. If "n" is omitted, the internal width is assumed to be the same as "w".

For output, the "n" characters are placed left-justified in the field of "w" columns. For input, the first "n" characters are taken from the field of "w" columns.

Hexadecimal

The control character for hexadecimal conversion is "X". The format term is given in the form

```
Xw.n
```

where "w" is the external field width, and "n" is the internal field width. If "w" is omitted, a default of 8 is assumed; if "n" is omitted, a default of (w+1)/2 is taken, rounded down to the nearest integer value.

For output, "n" internal bytes are unpacked and placed into "w" columns of the output line image. For input, "w" digits are packed and placed right-justified into "n" bytes.

Packed Decimal

The control character for packed decimal conversion is "P". The format term is given in the form

```
Pw.n
```

where "w" is the external field width and "n" in the internal field width. If "w" is omitted, a default of 8 is assumed.

For output, "n" bytes are unpacked and placed into "w" columns in the output line image. If "n" is omitted, standard format is used (see the subsection "Standard Format I/O"). For input, "w" digits from the input line image are packed and placed into "n" bytes.

Literals

It is often desirable to place labels and titles into the output line image directly from the format. This can be done by enclosing the material to be "transferred literally" to the output line in either primes (') or double-quotes (").

For example, when printing the result for the factorial example given above, the result could be labeled as follows:

```
          PRFMT  FMT2,(RESULT,0)
           .
           .
     FMT2  DC     C'" RESULT = ",I10*'
```

This format would print the result in the form

```
     RESULT =        120
```

Note that for integer conversion, the number is placed right-justified in the output line field.

If the literal string is initiated with a prime, it must be terminated with a prime. Likewise, if the string is initiated with a double-quote, it must be terminated with a double-quote.

The first character of each record of printed output is usually interpreted as a carriage-control character for controlling the positioning of the paper in the printer. It is convenient to use literal format items to insert this character. The first character of the output line produced by FMT2 in the previous example is a blank which causes single spacing (a single line is skipped before the output line is printed). The format

```
"-",2F3.3*
```

will cause triple spacing (three lines are skipped before the line containing two floating-point numbers is printed). A list of legal carriage-control characters is given in the Appendix H to the section "Files and Devices" in MTS Volume 1, The Michigan Terminal System. The most commonly used are:

```
blank  - single space
  0    - double space
  -    - triple space
  1    - skip to top of next page
  2    - skip to next half page
  4    - skip to next quarter page
```

Spaces and Tabs

It is often desirable to space numbers in the output line and to provide "tabulator stops". One method of spacing numbers is to give large field widths; because the numbers are right-justified, spacing is provided as such. However, control characters can be used for explicit skips and transfers.

Spaces

The control character "S" is used to specify the number of columns to be spaced. The format term is given as

```
Sn
```

where "n" specifies the number of columns to be spaced. "n" may be either a positive or negative number. For example, S20 will space twenty columns.

For output, the line image may be considered similar to a line being typed on a typewriter. At any instance, the carriage is sitting at the next column to be used. The S format term is equivalent to hitting the space bar "n" times (if "n" is positive) or hitting the backspace key "n" times (if "n" is negative).

For input, a similar analogy may be used. The S format term may be used to space forwards or backwards over the input line image.

Tabs

The control character "T" is used to tab (transfer) to a particular column in the input or output line image. The format term is given as

Tn

where "n" is the column number to tab to. The typewriter analogy falls apart a little here, because in a typewriter the tab only goes forward, whereas here the "carriage" can be tabulated either forwards or backwards. For example, T20 causes the next item to start in column 20, regardless of where the carriage currently is.

Note that nothing is done to the columns that are spaced or tabbed over. If they contained information, it still remains; if they contained blanks, they remain blank.

Line Skips

It is often desirable to print more than one line or read more than one input line with a single format. For this reason, the control character "/" (slash) is provided. For output, whenever a slash is encountered, the current line is printed, and a new line is started, beginning in column 1. For example, to skip to a new page, print a floating-point number, double space, and print another number, the format could be:

```
"1",F10.9/"0",F5.3*
```

For input, whenever a slash is encountered, a new input line is read starting at column 1. For example,

```
I3/6F4.2.8*
```

reads a 3-column integer from one line and six 8-column floating-point numbers from the following line.

Modifiers

Modifiers may be used to alter the effect of input or output conversion. For example, the two modifiers discussed earlier, H and M, specify that conversion is to take place from or into a halfword and that the decimal point is to be suppressed for F output conversion, respectively.

A modifier may be specified in the form of a letter, a special character, or a format variable. The modifier must always appear with the control character. A modifier that does not require a count may appear anywhere in the format term, for example,

RC8.6 or CR8.6

specifies that a 6-byte character string is to be right-justified into an 8-column field on output or that the rightmost 6 bytes of an 8-column character field are to be placed into 6 bytes on input. A modifier that does require a count must appear after the control character and must be separated from the preceding fields with a colon ":". For example,

C8.6:A20

specifies on output that the 8-byte character string is to be centered at column 20. If there are two modifiers requiring counts, they must be separated by colons, for example,

I4:G8:A16

specifies that an integer is to be converted using octal base conversion and placed into a 4-column output field that is centered at column 16.

The table below gives the modifiers that are available for IOH. The complete description of each of these modifiers is given in Appendix A to this section.

Modifier	Function
A	Centering control (output only)
B	Byte conversion
D	Doubleword conversion
G	Base conversion
H	Halfword conversion
J	Ignore field width (output only)
L	Left justification
M	Suppress decimal point
N	Null fill character (input only)
R	Right justification
U	Fill, if zero
W	Fullword conversion
Y	Forced plus sign
Z	Zeros fill character
$	Floating dollar sign (output only)
@	Scale factor (output only)
¬	Packed decimal sign

Multiplicities, Groups, and Blocks

If the same format term is to be repeated several times, a multiplicity factor may be used. For example,

```
I10,I10,I10
```

may be written more compactly as

```
3I10
```

where the integer 3 (the multiplicity) means that the following specification is to be taken three times. In this case, one format term will match three items in the list. Whenever there is no integer preceding the control character, a multiplicity of 1 is assumed.

It is often desirable to repeat groups of format terms. This can be done by enclosing the group in parentheses and placing the multiplicity in front of the left parenthesis. For example,

```
E5.3,E5.3,E5.3,I2,F5.1,F5.1,F5.1,I2,F5.1,F5.1,F5.1,S10,I10*
```

may be written more compactly as

```
3E5.3,2(I2,3F5.1),S10,I10*
```

Such grouping parentheses can be nested as deeply as desired.

A multiplicity of zero in front of a format term or a group means "do it zero times", i.e., "do not do it at all". Therefore, 0F5.5.11 will do nothing and

```
S10,0(S10,F5.5.11,'TRA '/),I3*
```

will skip 10 columns and print a 3-column integer; nothing inside of the parentheses will be printed. This is most useful where there is some kind of format-variable, rather than an explicit zero multiplicity in front of the left parenthesis.

A range of locations may be specified by using a block specification. A block is given in the form

```
x,...,y
```

where "x" and "y" are the first and last elements in the range, respectively. The following statement may be used to print 5 integers which occupy consecutive locations in storage, each occupying a fullword:

```
PRFMT  FMT3,(B,...,B+4*4,0)
```

(the reason for the "*4" is that each integer occupies 4 bytes of storage). Note that this is equivalent to specifying five items in the list. The format FMT3 could be

```
FMT3  DC   C'5I*'
```

The list elements are printed, starting with the first one and going in the proper direction through and including the last one, B+4*4. The address of the last element need not be higher than the address of the first element in a block specification, i.e., the block may specify that the items are to be taken in reverse order. The above block could have been specified as

```
PRFMT  FMT3,(B+4*4,...,B,0)
```

Format-Break Characters

Format terms may be separated either explicitly or implicitly. When the format terms are separated explicitly, a format-break character, is used to separate the terms. The format-break character may be either a comma or a matched pair of parentheses, for example,

```
I10,I5  or  I10(I5)
```

When the format terms are separated implicitly, an implicit comma is inserted into the format by IOH. The format terminator "*" has an implicit comma to its left, e.g.,

```
I5*
```

is equivalent to

```
I5,*
```

The characters "/" (line terminator), "|" (line terminator), "?" (parameter list terminator), "#" (default indicator), and literal fields have implicit commas both to their left and to their right. Because implicit commas exist for these positions, it is not necessary to explicitly insert a comma in those places, e.g., it is not necessary to insert a comma before a literal field. However, when a multiplicity factor is used with a literal field, for example,

```
F5.3,3('HEADING ')
```

then an explicit comma must appear to separate the format correctly.

Rescanning the Format

It has been stated that the list consists of a set of names of variables to which or from which information is to flow, and that each list item matches a field specification in the format. What happens if (1) the list has more items than the format has fields, or (2) the list has fewer items than the format has fields?

In all cases, all the list items must be accounted for. During execution, IOH scans the format. When it finds a field specification, it goes to the list to determine where to place (for output) or where to get (for input) the information that is to be transferred. If IOH does not find a list item, because they have all been "used up", it stops at that point, just as if it had encountered an asterisk while scanning the format.

If IOH reaches the end of the format (as designated by the asterisk), it looks at the list to determine if there are any items on the list that have not been "used up". If the list is exhausted, normal format termination occurs, i.e., if this was an output format, the line is printed, and control returns to the user's program. If the list is not exhausted, the line is printed (if output) or a new line is read (if input) and the format is rescanned. If no parentheses are used in the format, the format is restarted from the beginning. For example, the format

```
10I8*
```

from the statement

```
RDFMT   FMT4,(BLOCK,...,BLOCK+100*4,0)
```

will read 101 integers from 11 lines; 10 integers will be read from each of the first 10 lines, and the 1 integer from the eleventh line. If this were a print statement, 11 lines would be printed, 10 integers on each of the first 10 lines and 1 integer on the eleventh line.

If there are parentheses used in the format, format scanning starts at the end of the format and works toward the front of the format, until a zero-level left parenthesis is found (one that is not inside another left parenthesis). Format scanning again starts forwards using any multiplicity that may be in front of this parenthesis. For example, the format

```
I3,F5.3,F10.3*
```

will read input lines containing an integer in columns 1-3 and floating-point numbers in columns 4-13 and 14-28 until the list is exhausted. The format

```
(I3,2(F5.3,F10.3))*
```

will read input lines containing an integer in columns 1-3 and floating-point numbers in columns 4-13, 14-28, 29-38, and 39-53. The format

```
I3,2(F5.3,F10.3)*
```

will read an input line containing an integer in columns 1-3 and floating-point numbers in columns 4-13, 14-28, 29-38, and 39-53, followed by input lines containing floating-point numbers in columns 1-10, 11-25, 26-35, and 36-50. And as a final example, the format

```
I3/2(F5.3,F10.3)*
```

will read an integer from columns 1-3 of the first input line, and all other list items from successive lines, four floating-point numbers per line. This type of format is often used to read a data count from the first line, and the data items from the remaining lines.

STANDARD FORMAT I/O

If a data-transmission format term (C, E, F, I, P, or X) consists of only the control character, i.e, the field widths and delimiting decimal points are omitted, standard format input or output is assumed. For input, the data items in the input line-image are in free format (not in any specific columns) and are separated by a character, e.g., a comma, that could not normally appear in the input field. For numeric output, the default field widths are used. For character output, the default field width of 1 is used, and the resulting character is placed between literal-break characters. Initially, a prime is used as the literal-break character, but in general, the "latest-used" literal-break character delimits standard format character output.

For standard format numeric input (E, F, I, P, and X), the input line-image is scanned for the next occurrence of a character that is not an input fill character (initially a blank). The first such character is set as the initiator character. After setting this character, the scan proceeds until the occurrence of the next comma, the next input fill character, or the end of the input line-image (any of which is set as the terminator character). All characters between the initiator character (including it) and the terminator character (excluding it) are taken as the input field. For example, the format term

```
3I
```

may be used with the input line

```
1,2,3
```

to read the values 1, 2, and 3. If the scan reaches the end of the input line-image before setting the initiator character, a new line-

image is requested (i.e., a call to CLOSE followed by a call to OPEN is made). If two consecutive commas appear in the input line-image, the input data item is taken as null, no conversion takes place, and the next argument in the parameter list is skipped. If one or more consecutive input data items are the same, the input data item may be enclosed in parentheses with the corresponding multiplicity inserted before the left parenthesis. For example, the input line

```
4,4,4,2.0,2.1,2.1
```

may be written as

```
3(4),2.0,2(2.1)
```

For standard format character input, the input line-image is scanned for the next occurrence of any nonblank character, except a comma. If such a nonblank character occurs, it is checked to see if it is a literal-break character. If it is not, an error condition results; if it is, the scan continues looking for the next occurrence of the same literal-break character. All characters between the initial literal-break character and the terminal literal-break character are taken as the input field. With standard format character input, a comma or an input fill character must be used to separate the data items. For example, the format term

```
2C
```

may be used with the input line

```
'ABCDEF','WXYZ'
```

to read the character strings "ABCDEF" and "WXYZ". If the input data item contains a literal-break character as part of the data item, two successive literal-break characters must be used in the data item (the first of the pair is used and the second is discarded). The scan then continues until a literal-break character not immediately followed by another literal-break character occurs. For example, the above format may be used with the input line

```
'ABC''123',''''456'
```

to read the character strings "ABC'123" and "'456". Warning: the blank and the comma should not be used as literal-break characters when standard format character input is being used.

As a more complex example, the input format

```
2I,2F,C*
```

may be used to read the following input lines:

```
10  23  6.4  5.9  'DATA 1'
4,  27,  3.2,  10.,  'DATA 2'
```

```
     2(4)  ,  2(3.2),  'DATA 3'
     6 ,,  3.2 ,,  'DATA 4'
```

In the last line, the second and fourth input items are null, and the corresponding arguments in the parameter list are skipped.

IOH MACROS

Since the calling sequence for IOH is rather complicated, a set of macro definitions is provided to allow the user to easily call IOH.

These macros may be subdivided into two groups; macros that may be used to initially call IOH and macros that may be used to continue the processing of an input or output line by IOH. The macros that may be used to initially call IOH are:

```
RDFMT  - Read formatted input (defaults to SCARDS)
WRFMT  - Write formatted output (defaults to SPRINT)
PCFMT  - Punch formatted output (defaults to SPUNCH)
PRFMT  - Print formatted output (defaults to SPRINT)
SERFMT - Print formatted output (defaults to SERCOM)
GUSFMT - Read formatted input (defaults to GUSER)
LKFMT  - Look at formatted input (defaults to SCARDS)
```

The macros that may be used to continue processing are:

```
MOREIO - Continue reading or writing formatted input or output
ONEIO  - Continue reading or writing formatted input or output one
         element at a time
ENDIO  - Terminate I/O processing
IOP    - I/O parameter (acts like ONEIO or MOREIO depending on the
         number of arguments given)
REFMTC - Repeat format call
ACCEPT - Close LKFMT buffer
IOPMOD - Set I/O FDname modifiers
```

The general form of the macro calls that may be used to initially call IOH is as follows (the RDFMT macro is as an example prototype):

```
RDFMT format,list,OPEN=subr,CLOSE=subr,EOF=subr,ERROR=subr,
      LUNIT=unit,SECT={sect|*},POOLSW={0|1},NC=name,TYPE={A|S},
      SYMTBL=table
```

where "format" is the user-supplied format and "list" is the list of data items to be used. The format specifies how to interpret fields being read or written. "format" is the label of the format statement. Whenever the term "list" is used in the following descriptions, it will have the following meaning:

The "list" is a list of arguments that may be either a single element, a list of elements, or null. Block arguments may also be specified, such as A(1),...,A(10). In this case, IOH will use all arguments from A(1) to A(10), inclusive. There may be no other arguments between A(1) and A(10). If there is more than one element in the list, the entire list must be enclosed in parentheses.

The keyword parameters in the macro call are optional. Any or all may be specified and they may appear in any order. For example,

```
PRFMT FMT1,(A,B,A+100,...,A+200,0),POOLSW=1,TYPE=S,CLOSE=MFCLOSE,
      OPEN=MLOPEN,SECT=PQSECT,NC=12
```

The optional keyword parameters are

OPEN=subr — "subr" specifies the name of a user-supplied subroutine that IOH will call when it needs to read a new line-image; this subroutine may be internal or external to the calling program; if it is external, the name must be defined by an EXTRN assembler language statement.

CLOSE=subr — "subr" specifies the name of a user-supplied subroutine that IOH will call when it needs to write a line-image (i.e., release a line-image); this subroutine may be internal or external to the calling program; if it is external, the name must be defined by an EXTRN assembler language statement.

Note: If the user is supplying either an OPEN or a CLOSE subroutine, he should supply both (especially when using the read format macro). See the description of OWNCONVR in the subsection "Additional Entry Points to IOH" for details of writing an OPEN or a CLOSE routine.

The user may specify that on each return from IOH, the return code should be checked. If either the EOF or ERROR parameters are specified, code is generated to check the return code and to route the return to the point specified by the user. Note: IOH returns to the user with a nonzero return code only if the correct flag bits have been previously set by a call to SETIOHER. See the description of SETIOHER in the subsection "Additional Entry Points to IOH."

EOF=subr — "subr" specifies the name of a user-defined or system-defined subroutine to handle end-of-file returns from IOH; the subroutine may be internal or external to the calling program; if it is external, it must be defined by an EXTRN assembler language statement. Note: IOH allows an end-of-file return only if the user has set the appropriate flag (hex 40) by calling SETIOHER; if this flag is not set, a

return is made to the system subroutine SYSTEM with the comment:

ALL INPUT DATA HAS BEEN PROCESSED AT LOCATION xxxxxx.

ERROR=subr "subr" specifies the name of a user-defined or system-defined subroutine to handle error returns from IOH; the subroutine may be internal or external to the calling program; if it is external, it must be defined by an EXTRN assembler language statement. Note: IOH allows an error return only if the user has set the appropriate flag (hex 80) by calling SETIOHER; if this flag is not set, a return is made to the system subroutine ERROR with the comment:

ERROR RETURN TO SYSTEM.

LUNIT=unit "unit" specifies the logical I/O unit number (between 0 and 9) to be used in reading or writing formatted input or output. If LUNIT is omitted, the subroutines given in the table below will be used for input and output. If LUNIT is specified, the logical I/O unit named (e.g., LUNIT=0) will be used to read in input (using the READ subroutine) or write output (using the WRITE subroutine). "unit" may also be the address of an FDUB-pointer; the line-image length in this case is 256.

SECT={sect|*} "sect" specifies the name of a control section or location counter in which the parameter lists generated by the current macro call are to be placed. The SECT parameter need not be given on every call. The effect of a previous SECT parameter is retained until the appearance of a new SECT parameter. Initially, macro calls are generated with in-line parameter lists. The user may specify that these parameter lists be placed in the control section or location counter designated by the last SECT argument. If SECT=* is specified, in-line parameter lists are again generated.

POOLSW={0|1} This parameter specifies whether halfword constants are to be regenerated. Each time a macro call is expanded, halfword constants are generated which form the counts for the lists and sublists. By default, halfword constants previously generated are <u>not</u> regenerated; the previously defined halfwords are used (i.e., the macro processor keeps track of labels assigned to halfword constants that <u>it</u> has generated previously). By using this parameter, the user may avoid generating redundant halfword constants; it is a space-saver in this respect. Each time POOLSW is

explicitly used (whether it is set to 0 or 1) in a macro call, all previous halfword constant reference accumulation is lost (i.e., the halfword constant pool is rebuilt from scratch). The initial value of POOLSW is 0 specifying that constants are not regenerated. If POOLSW is set to 1, then new constant pools are generated for each macro call thereafter until it is reset to 0.

NC=name — The NC parameter used to assign names to macro calls. The user may then refer the assigned name in a later macro call using the same format and list defined in the named macro call. This allows the user to use the same format and parameter list for many macro calls; it is a space-saver. "name" may be any string of characters up to eight characters in length. Only twenty-five such (different) names may be used in a single assembly. When the NC parameter is specified in a macro call, a search is made of a table that contains the names of the previously defined macro calls using the NC parameter. If the NC parameter name is not found, then an entry for the macro call is made in the table and processing proceeds as if the NC parameter was omitted. If the NC parameter is found in the table, then both the "format" and the "list" of the present macro call are ignored and the "format" and the "list" of the macro call that originally defined the present name are used. When the NC parameter is used in a REFMTC macro call, it specifies that, in addition to using the "list" and "format" of a previous macro call, the OPEN and CLOSE arguments are also to be used (see the REFMTC description below).

TYPE={A|S} — This parameter specifies the type of adcon lists to be generated. The lists that the macro processor generates may be in one of two types. The first type consists of fullword adcons (A-type adcons) and the second type consists of halfword adcons (S-type adcons). Initially, an A-type list is generated. TYPE may be set to either A or S, to generate A-type adcons and S-type adcons, respectively. The type selected is used in all succeeding macro calls until the TYPE is changed.

SYMTBL=table — "table" specifies the address of the symbol table to be used with format variable input and output. See the subsection "Format Variables" for further details and examples using format variables.

If the OPEN, CLOSE, and LUNIT parameters are omitted, the macros will generate adcons with the names of the appropriate system-supplied OPEN and CLOSE routines. They are as follows:

Macro Call	OPEN	CLOSE	Routine Called
RDFMT	ROPEN	RCLOSE	SCARDS
WRFMT	POPEN	PCLOSE	SPRINT
PCFMT	PCOPEN	PCCLOSE	SPUNCH
PRFMT	POPEN	PCLOSE	SPRINT
SERFMT	SEROPEN	SERCLOSE	SERCOM
GUSFMT	GOPEN	GCLOSE	GUSER
LKFMT	LOPEN	LCLOSE	SCARDS

Notes:

(1) PRFMT is normally used for writing output to a line printer. WRFMT is normally used for writing output to any I/O device capable of receiving the line-image. PRFMT and WRFMT actually generate the same calls.

(2) A call to RCLOSE causes the input line-image transmitted by a previous call to LOPEN or ROPEN to be closed.

(3) The line length is dependent upon the file or device attached.

Whenever a zero is used as an argument, I/O processing is terminated (the line is closed and a return is made to the caller). The zero element may appear anywhere in the argument list. If the argument list is exhausted, all translations are completed, and no zero element is found, control returns to the caller. The user has the option of returning to IOH with another argument list and/or a new format. If he does not return or if he makes a new initial call to IOH (the initial types of call have been defined above), the remainder of the previous line-image is discarded. If he returns with another parameter list, I/O processing continues with the previous format and line-image. This process can continue until a zero element is encountered or until no return is made.

The return to IOH to continue processing of the input or output line can be made with any of the following macros:

MOREIO The MOREIO macro is used to specify a new format or a new argument list, or to specify the EOF, ERROR, SECT, NC, POOLSW, or TYPE parameters. The general form of the call is

```
MOREIO list,EOF=subr,ERROR=subr,SECT={sect|*},CFMT=format
       POOLSW={0|1},NC=name,TYPE={A|S}
```

If CFMT is specified, the information about the previous format is discarded (i.e., the new format is used as if it were the original one). In particular, information about the placement of parentheses and their associated multiplicities in the previous format is discarded. With the CFMT parameter, the user may build a format during program execution (only part of the format need be present at any one time). The user should remember to supply enough of the format each time for

the number of arguments in the associated parameter list or to to terminate each such format with an asterisk "*". If the format is terminated with an asterisk, format scanning resumes at the beginning of the format, or with the last level of multiplicity. If the NC parameter refers to a previously macro call, then the "list" and CFMT arguments of the present call are ignored and the "list" and "format" (or CFMT "format") of the previous macro call are used.

ONEIO The ONEIO macro is used to specify one list item at a time during input or output. The EOF and ERROR arguments may also be specified. The general form of the call is:

```
ONEIO arg,EOF=subr,ERROR=subr
```

"arg" is the list item to be used. If "arg" is zero (i.e., ONEIO 0), I/O processing is terminated. If "arg" is omitted, general register 1 is assumed to contain the address of the list item.

ENDIO The ENDIO macro is used to terminate I/O processing. This is equivalent to ONEIO 0. The general form of the call is:

```
ENDIO
```

IOP The IOP macro is used to specify either a single list item or a single blocked-pair list item. The EOF and ERROR parameters may be used in the macro call. The general form of the call is:

```
IOP arg1[,...,arg2],EOF=subr,ERROR=subr

IOP arg
```

is equivalent to

```
ONEIO arg
```

and

```
IOP arg1,...,arg2
```

is equivalent to

```
MOREIO (arg1,...,arg2)
```

REFMTC The REFMTC macro is used to "execute" previously defined initial-type macro calls. The general form of the call is:

```
REFMTC NC=name,ERROR=subr,EOF=subr
```

The NC parameter must be present and refers to a previously defined initial-type macro call using that NC parameter. The

effect of this type of call is to generate code which uses the OPEN and CLOSE parameters of the initial-type call (even if they are defaulted), in addition to using the "format" and "list" of that call. The use of this macro call would, for example, allow the user to read an input line from several different places in a program using the exact same "format", "list", and OPEN and CLOSE options. It is a space-saver in this respect.

ACCEPT The ACCEPT macro is used to close the input buffer that is being "looked at" by a LKFMT call. The general form of the call is

```
ACCEPT LUNIT=unit
```

The LUNIT parameter specifies the logical unit that the input buffer was read from; if this parameter is omitted, SCARDS is assumed.

The IOPMOD macro may be used to call the IOH entry point IOPMOD. This allows the user to specify the modifiers and the line number that will be used by the IOH OPEN and CLOSE routines when such routines call the system subroutines READ, WRITE, SCARDS, SPRINT, SPUNCH, SERCOM, or GUSER. The IOPMOD entry point is described in detail in the subsection "Additional Entry Points to IOH."

The LKFMT macro call is used to "look at" an input line. When the LKFMT macro is invoked, the current input line that is in the input buffer is processed by IOH according to the specified format; if the input buffer is empty, a new input line is read into the buffer. The ACCEPT macro call is used to close the input buffer. After the input buffer is closed, another LKFMT call will read a new input line into the buffer. The LKFMT and ACCEPT macro calls are most useful when an input line is to be scanned under several different formats. For example, the following program segment illustrates how LKFMT can be used to read an input field both as a character string and as a floating-point number.

```
          LKFMT  FMT1,(CHAR,0)
          LKFMT  FMT2,(FLT,0)
          ACCEPT
           .
           .
CHAR      DS     CL10
FLT       DS     E
FMT1      DC     C'C10*'
FMT2      DC     C'F5.3.10*'
```

The first 10 columns of the input line will be read into CHAR as a character string and into FLT as a floating-point number.

To avoid conflicts between user-program names and names generated during a macro call, all names generated during macro processing begin with a pound sign "#" and have an identifier immediately following the

pound sign; The first character of the identifier is numeric. All generated names are at least three characters in length. All global set-symbols used during macro call compilation begin with a pound sign. The names of the macros that are used in expanding the IOH macros described above are:

```
#IOHPRLS
#IOHOCCK
#IOHERCK
```

Thus, the user should not define his own macros with any of the above names.

The following example illustrates the use of macro calls.

```
PRFMT FORMAT1,(A,B,C,...,C+8,A+5,A+10,...,A+15,0),OPEN=IOPEN,
      CLOSE=ICLOSE
```

In this example, two user-defined input and output are specified. The above example may also be specified using the following set of macro calls:

```
PRFMT FORMAT1,OPEN=IOPEN,CLOSE=ICLOSE
IOP A
IOP B
IOP C,...,C+8
IOP A+5
IOP A+10,...,A+15
ENDIO
```

SPECIAL FEATURES OF IOH

Additional Entry Points to IOH

SETFRVAR

General register 1 points to the beginning of a format variable vector. The beginning of the vector must be halfword-aligned. See the subsection "Format Variables" for further details on the use of format variable vectors.

SETIOHER

General register 1 points to a 4-byte area (which need not be halfword- or fullword-aligned) that contains flags to specify actions to be taken in special situations. These flags are ORed with the flags that are currently set. The flags in the first byte are:

Bit 0 (Hex 80) If set, an error return is to user.
If not set, an error return is to the system (the default).

Bit 1 (hex 40) If set, an end-of-file return is to user.
If not set, end-of-file return is to the system (the default).

Bit 2 (hex 20) If set, full recovery is attempted after an error; I/O processing will attempt to continue. Note: Bit 0 or bit 3 must also be set.
If not set, no error recovery is attempted (the default); I/O processing is terminated.

Bit 3 (hex 10) If set, error comments via SERCOM are suppressed. Note: Bit 0 must also be set.
If not set, error comments are printed (the default).

Bit 4 (hex 08) If set, a new line will start if a converted output line exceeds the maximum length (note: an error is still flagged if a T format item caused the error.) Note: Bit 0 or bit 3 must also be set.
If not set, an error will result if a converted output line exceeds the maximum length (the default).

Bit 5 (hex 04) If set and a field width exceeded error occurs on output, the output field will be filled with the overflow character and processing will continue (the default).
If not set, an error condition is flagged.

Bit 6 (hex 02) If set, extended error messages are produced (default for batch mode).
If not set, abbreviated error messages are produced (default for conversational mode).

Currently, none of the other bits in this 4-byte area are used.

DROPIOER

General register 1 points to the 4-byte area that corresponds to the 4-byte area of SETIOHER. Any bit that is set by a call to DROPIOERR causes the corresponding flag bit in the SETIOHER area to be reset to zero. Thus, the 4-byte area used to set the flags by a call to SETIOHER can be be used to reset them by a call to DROPIOER.

The following program segment illustrates how calls to SETIOHER and DROPIOER may be used to control end-of-file processing by IOH. In this example, input lines are read until an end-of-file is

encountered, at which point an exit is made to another segment of the program which resets the end-of-file exit.

```
                LA    1,SETAREA                 Enable EOF exit
                L     15,=V(SETIOHER)
                BALR  14,15
        DATARD  RDFMT FMT,(INT,0),EOF=DATAEND   Read input data
                 .
                 .
                B     DATARD
        DATAEND LA    1,DRPAREA                 Disable EOF exit
                L     15,=V(DROPIOER)
                BALR  14,15
                 .
                 .
        SETAREA DC    XL4'40000000'             Specify EOF flag
        DRPAREA DC    XL4'40000000'
        FMT     DC    C'I6*'                    Format
        INT     DS    F
```

GETIOHER

General register 1 points to a 4-byte area into which the 4-byte area of the last call to SETIOHER is moved. Using this, the user can determine which bits are currently set.

OWNCONVR

With the O format term, the user may use his own conversion routine for either input or output conversion. Before using such a conversion routine, the user must specify the entry point address of his routine. The calling sequence

```
LA   1,myconv
L    15,=V(OWNCONVR)
BALR 14,15
```

or

```
LA   1,myconv
CALL OWNCONV
```

specifies "myconv" as the entry point address. The routine "myconv" need only save general register 14 (so that it can return). If and when the return is made, general register 15 should be 0 if the conversion was successful, 4 if an end-of-file condition was encountered on the call to the OPEN routine, or 8 if the conversion was unsuccessful. All other codes are illegal and will be treated as errors. When IOH calls such a routine, general register 1 points to the beginning of an area which has the following layout:

```
OWNW1     DS   F
OWNW2     DS   F
OWNW3     DS   F
OWNARG    DS   F
OWNOPEN   DS   F
OWNCLOSE  DS   F
OWNCUR    DS   F
OWNLAS    DS   F
OWNPUT    DS   F
OWNFLAGS  DS   XL2
```

where

OWNW1 - first field width (e.g., the "m" in Em.n.w).
OWNW2 - second field width (e.g., the "n" in Em.n.w).
OWNW3 - third field width (e.g., the "w" is Em.n.w).
OWNARG - the address of the conversion argument (i.e., from or to where the conversion is to take place).
OWNOPEN - the address of the routine that the user's routine should call if he wishes to get a new input or output line-image.
OWNCLOSE - the address of the routine that the user's conversion routine should call if he wishes to close out the input or output line-image.
OWNCUR - the pointer to the current byte in the input or output line-image.
OWNLAS - the pointer to the last byte in the input or output line-image.
OWNPUT - the address of a routine that the user may call to place an item into the output line-image.
OWNFLAGS - two bytes of flag bits:

byte 1: bit 0 - set if R was specified
1 - set if L was specified
2 - set if N was specified
3 - set if DD was specified
4 - set if D was specified
5 - set if W was specified
6 - set if H was specified
7 - set if B was specified

Only <u>one</u> of bits 3,4,5,6, or 7 may be set.

byte 2: bit 0 - set if standard format was specified
(i.e., no 0's or field widths)
1 - set if field width 1 was a blank
2 - set if field width 2 was a blank
3 - set if field width 3 was a blank
4 - set if field width 3 was found
5 - set if any period "." was found
6 - set if for output, not set for

input

7 - set if this was the first time through this format term

Note:

(1) On input, the user will get the same input line-image as on the previous call, if he does not call OWNCLOSE before calling OWNOPEN. Thus, the user may rescan a line-image (starting at column 1) by calling OWNOPEN without calling OWNCLOSE for the previous line-image.

(2) The user may change any of the fields specified.

On input:

The user should do all of his own conversion into an area specified in OWNARG; on return, OWNCUR should contain the address of the byte to which the line-pointer is to point in the input line-image.

On output:

The user has two options:

(1) The user may move his conversion into the output line-image starting at the address in OWNCUR and not going past the address in OWNLAS for each line-image. He may use successive line-images. Each time a new line-image is obtained, OWNCUR and OWNLAS are changed appropriately.

(2) The user may call OWNPUT to place his output into the output line-image and thus take advantage of the L, R, Y, Z, and $ modifiers and the default fill character. When OWNPUT is called, register 1 points to a list of 3 adcons: the first adcon points to the beginning of the user's conversion output area; the second adcon points to the fullword number of bytes in that area; the third adcon points to the fullword conversion width. Adcon three must be greater than or equal to adcon two, or an error will result.

IOPMOD

The entry point IOPMOD allows the user to specify the I/O modifier bits and line numbers that will be used by the IOH OPEN and CLOSE routines when they call the system subroutines READ, WRITE, SCARDS, SPRINT, SPUNCH, SERCOM, and GUSER. The calling sequence using the IOPMOD macro is

```
IOPMOD unit,mod,lnum,MF=mf
```

or, alternatively, using the CALL macro

```
CALL IOPMOD,(mod,unit,lnum),MF=mf
```

"unit" is either the name of the location of an FDUB-pointer or fullword logical I/O unit number, a logical I/O unit number, or a logical I/O unit name enclosed in primes. "mod" (optional) is a parenthesized list of I/O FDname modifiers or the name of the location of a fullword containing the I/O FDname modifiers to be used in I/O calls by IOH. "lnum" (optional) is the line number or the name of a fullword containing the line number to be used; the line number is specified in its internal format (external-format x 1000). MF (optional) specifies special calling sequence generation formats (see the CALL macro description in MTS Volume 14, 360/370 Assemblers in MTS, for details).

The following example illustrates a call to IOPMOD which causes logical carriage-control to be suppressed for output written to SPRINT by IOH. This example is first given in assembly language and then in FORTRAN:

```
IOPMOD 'SPRINT',(@NOCC)

INTEGER*4 UNIT(2)/'SPRI','NT  '/,MOD/64/
CALL IOPMOD(MOD,UNIT,0)
```

Format Variables

The use of a format variable allows a program calling IOH to substitute the value of a variable into a format wherever a number would otherwise occur. This substitution is made at the time the format variable is encountered during the scan of the format by IOH. The less-than sign "<" begins a format variable and the greater-than sign ">" terminates a format variable. The format variable may be a single format variable name or an expression containing several format variable names. For example,

```
<VAR1>
```

defines the format variable name VAR1.

The format variable names are looked up in a user-defined symbol table and their appropriate values are substituted in place of the format variable in the format expression. For example, if VAR1 has the value 3, then the format expression

```
<VAR1>I
```

would become, after substitution

3I

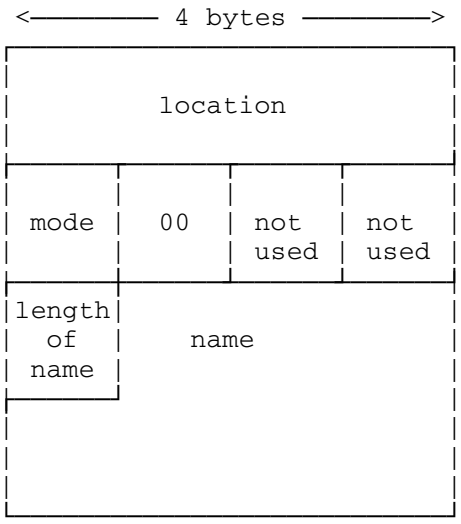
The address of the user-defined symbol table is specified in the macro call to IOH by the SYMTBL keyword, e.g,

```
RDFMT  format,list,SYMTBL=table
PRFMT  format,list,SYMTBL=table
```

"table" is the fullword address of the user-defined symbol table. This symbol table has the following format:

(1) the first word is a fullword count of the number of symbol entries.

(2) the second and following words are the symbol entries in the form:



"location" is the address of the value that is to be substituted for the format variable name in the format. "mode" indicates the length and data type of the value to be substituted; mode may be one of the following:

```
  0 - short-precision floating point (converted to integer)
1-4 - fullword integer
  5 - byte value
  6 - halfword integer
  7 - byte value
```

"length of name" is the number of characters in the symbol name (not including blanks); the name is padded with blanks so that each entry is fullword aligned.

The format variable may be of any of the above modes, but the resulting value of the format variable must be between -32768 and 32767.

When using a format variable as a multiplicity, varying the multiplicity does not vary the number of items in the list. If it is necessary to skip items, format terms with zero external field widths may be used (see the example below).

The following short program segment illustrates the use of format variables to read a list of variable-width integers into an array. Both the macros RDFMT and MOREIO are used to illustrate how a single input line may be read by two separate I/O calls.

```
            RDFMT  FMT1,(M,W),SYMBTL=TABLE Read mult. and width
            LA     1,20              Compute remainder
            S      1,M
            ST     1,REM
            MOREIO (ARRAY,...,ARRAY+4*19,CHARS,0)
             .                       Read integers and string
             .
ARRAY       DS     20F               Array of integers
CHARS       DS     CL32                Character string
M           DS     F                 Number of integers read
W           DS     F                 External field width
REM         DS     F                 20 - number of integers read
FMT1        DC     C'2I5,<VARM>I<VARW>,<REM>I0,C*' Format
TABLE       DC     F'3'              Number of entries
ENTRY1      DC     A(M)              Address of M
            DC     X'01'             Mode is fullword integer
            DC     X'00'
            DS     XL2
            DC     X'04'             Length of name
            DC     C'VARM   '        Format variable name
ENTRY2      DC     A(W)              Address of W
            DC     X'01'             Mode is fullword integer
            DC     X'00'
            DS     XL2
            DC     X'04'             Length of name
            DC     C'VARW   '        Format variable name
ENTRY3      DC     A(REM)            Address of REM
            DC     X'01'             Mode is fullword integer
            DC     X'00'
            DS     XL2
            DC     X'03'             Length of name
            DC     C'REM    '        Format variable name
```

If the multiplicity and field width is to be 10 and 4, then execution of the program segment will read 10 integers of external field width 4 and place them into ARRAY; the last 10 elements of ARRAY will be set to zero by the I0 format term. After the integers are read, the string "DATA STRING" is read using standard format character input. The input data line would be set up as

```
   10   4   1   2   3   4   5   6   7   8   9  10 'DATA STRING'
|
└Column 1
```

The second format term, whose value will be 14I0, is needed to exhaust to list of input data items for ARRAY, i.e., zero-length integers are read into the last 14 elements of the array.

A second type of format variable is the list format variable which is specified by a V in the format. Whenever a V is encountered in a format term, IOH assumes that the next data item in the list points to a halfword integer, the contents of which is substituted into the format in place of the V. For each V used in the format, there must be a corresponding data item in the list. For example, if the next two list items point to halfwords containing 2 and 6, respectively, then the format term

```
VIV
```

is equivalent to

```
2I6
```

The following program segment illustrates how the above example may be set up.

```
          PRFMT  FMT2,(M,W,INT1,INT2,0)
           .
           .
M         DC     H'2'
W         DC     H'6'
INT1      DC     F'12345'
INT2      DC     F'67890'
FMT2      DC     C'VIV*'
```

This will print INT1 and INT2 in the following format:

```
 12345 67890
```

A third type of format variable is the vector index format variable which is given in the form

```
n=
```

where "n" is an index into a "user-defined" vector of halfword integers. When IOH encounters a vector index format variable in a format term, the index "n" is used to index the "n"th integer in the vector; this integer is then substituted into the format in place of the format variable. The index must be greater than zero; if the index is omitted, an index of 1 is assumed. For example, if the first, second, and third entries of the vector are 3, 2, and 4, respectively, then the format term

```
1=F2=.3=.10
```

is equivalent, after substitution, to the format term

3F2.4.10

Vector index format variable processing must be initialized by the calling the IOH entry point SETFRVAR. This can be done with the following type of call:

```
LA    1,vector
L     15,=V(SETFRVAR)
BALR  14,15
```

where "vector" is the beginning of the vector of halfword integers. The following short program segment illustrates how the above example might be set up.

```
            LA    1,VECTOR
            L     15,=V(SETFRVAR)
            BALR  14,15
            PRFMT FMT3,(A,B,C,0)
             .
             .
A           DC    E'1'
B           DC    E'2'
C           DC    E'3'
VECTOR      DC    H'3'
            DC    H'2'
            DC    H'4'
FMT3        DC    C'1=F2=.3=.10*'
```

This will print A, B, and C in the following format:

```
    1.0000    2.0000    3.0000
```

Changing Defaults

The user may change the default field widths and modes initially set by IOH and may change the interpretation that IOH makes for any of the characters in a format term. For example, the user may change the literal-break character to another character (e.g., $).

The command to change defaults must be delimited by the pound sign "#" (the default indicator). There are two modes of changing defaults: normal mode and keyword mode.

Normal Mode

The user may change certain default internal and external field widths for C, E, F, I, P, and X format terms. The following table indicates which field width defaults may be changed.

Conversion Type	Changeable Defaults
Cw.n	"w" and "n"
Em.m.w	"m" and "n" only
Fm.n.w	"m" and "n" only
Iw	"w"
Pw.n	"w" only
Xw.n	"w" only

The defaults are changed in the following manner. If any field widths are explicitly present in the format term, only those corresponding defaults are set to the values given in the format term (the other defaults are unchanged). For example,

```
#E2#
```

sets the default "m" field width value to 2, and

```
#F3.3#
```

sets both the default "m" and "n" field width values to 3. If both the field widths and the decimal points are omitted from the format term, all of the default field widths are reset to their initial values. For example,

```
#E#
```

resets the default "m" and "n" field widths to their initial values of 1 and 3, respectively.

The default mode for E, F, and I format terms may be changed by specifying the desired mode using the length modifier (i.e., B, D, H, W) with the format term. For example,

```
#DE#
```

sets the default mode for E conversion to doubleword and resets the default "m" and "n" values to 1 and 3, respectively, and

```
#WI10#
```

resets the default mode for I conversion to fullword and sets the default "w" value to 10.

Keyword Mode

Keyword mode is used to change the default values for input and output fill characters, literal-break characters, and other IOH parameters. Keyword mode is entered when the keyword indicator is specified (initially an ampersand "&"). The keyword is normally terminated by an equal sign "=" (the exception is PUSH and POP); the terminator may not be changed as it is treated as a part of the keyword. A variable length operand may follow the equal sign for certain keywords. The keywords are not translated by IOH (except to be translated so that all characters are uppercase); thus, the keywords must always appear as shown in this section (e.g., if the EXCHANGE keyword is used to exchange the interpretations of A and B, the BREAK keyword spelling would not change to AREBK).

The following is a list of keywords that may be used to change default values. Initially, upper- and lowercase letters have equivalent interpretations; however, they may be set such that, for example, "a" has a different interpretation than "A". The pound sign "#" is always interpreted as the default indicator; however, other characters may also take on this function.

OUTFILL=x — change the current output fill character to the character "x". For example, the following format changes the output fill character to "@".

```
#&OUTFILL=@#
```

INFILL=x — change the current input fill character to the character "x". For example, the following format changes the input fill character to the character zero "0".

```
#&INFILL=0#
```

OVCHR=x — change the overflow fill character to the character "x". The overflow fill character (initially an asterisk) fills the entire field on output when the field width has been exceeded and when the bit is set (done on a call to SETIOHER) which allows such filling to occur. For example, the following format sets the overflow fill character to the dollar sign.

```
#&OVCHR=$#
```

BREAK=x — interpret the character "x" as a literal-break character. The user may have any number of literal-break characters. For example, the following format allows the dollar sign to be interpreted as a literal-break character.

```
#&BREAK=$#
```

EQUIV=xy interpret the first character "x" the same as the second character "y". For example, the following format allows an A in a format to have the same interpretation as a B (e.g., as the byte modifier).

```
#&EQUIV=AB#
```

EXCHANGE=xy exchange the interpretation of the two characters "x" and "y". For example, after processing the following format, D specifies character conversion and C specifies the doubleword modifier (assuming that their interpretations have not been changed previously).

```
#&EXCHANGE=DC#
```

REPLACE=xy the first character "x" is given the interpretation of the second character "y" and the interpretation of "y" is made null (i.e., illegal). For example, after processing the following format, C specifies E-type conversion and E is illegal (assuming that their interpretations have not been changed previously).

```
#&REPLACE=CE#
```

If both characters are identical, then the interpretation of that character is made null.

MODE={BCD|EBCDIC} specifies either BCD or EBCDIC. Initially, MODE is set to EBCDIC. The MODE setting determines whether plus signs, when put into the output line-image in a numeric conversion, will be either in BCD-mode (a 12-punch) or in EBCDIC-mode (a 12-6-8-punch). IOH accepts either the BCD or EBCDIC representations of the plus sign on <u>input</u> regardless of the mode setting. For example, after processing the following format, all plus signs produced on a numeric conversion by IOH will be in BCD-mode.

```
#&MODE=BCD#
```

{PUSH|POP} transfer from one default level to another. PUSH is used to transfer to a new default level. After this transfer, the user has at this level all the default values as they are initially given. given. POP is used to return to a previous default level. After this return, the user regains the previous default values and the default values from the level from which he returned are lost. This allows the user to have one interpretation for A at one level and another interpretation for A at a different level. For example,

```
#&BREAK=$&REPLACE=''&PUSH&BREAK=?#
```

makes the dollar sign the only literal-break character at the first level while both the dollar sign and the question mark are legal literal-break characters at the second level; &REPLACE='' has made the prime an illegal character at the first level.

POPALL — return the user to level 0 with respect to the default levels.

STATUS — the next item in the parameter list is assumed to point to a halfword into which the present default level (PUSH-POP) level number is placed. The initial default level number is 0.

RESET={key|x} — reset default values at the present default level. The defaults that may be reset to their initial conditions are specified by the second parameter "key". "key" may be one of the following:

- ALL - all characters will have their initial interpretations, and all default field widths and modes for all data-transmission conversion will have their initial values.
- TABLE - only the interpretations of the format characters will be reset.
- DEFAULT - only the default field widths and modes for data-transmission terms will be reset.
- BREAK - the literal-break characters are reset to the prime and double quote, and all other current literal-break characters are reset to their initial interpretations.
- OUTFILL - the output fill character is reset to a blank.
- INFILL - the input fill character is reset to a blank.
- OVCHR - the overflow fill character is reset to an asterisk.
- MODE - the mode is reset to EBCDIC.

If "key" is not specified, the character "x" is given its initial interpretation.

APPENDIX A: IOH FORMAT SPECIFICATIONS

This appendix contains the complete descriptions of the format control characters and modifiers that may appear in an IOH format.

A Centering Control

The centering control character is used to indicate in which column in the output line-image the converted field is to be centered. Thus, with the format

F1.4.10:A14

if the converted number were

▪▪▪2.1000

(where "▪" represents a blank), the decimal digit 2 in the converted number would appear in column 13 with three blanks before it and the characters ".1000" after it.

If "n" represents the number found after the A in the format term, and if "w" represents the total field width in the line-image, then the beginning of a field with field width "w" and centering in column "n" will start in column "n-[w/2]+1", rounded down to the nearest integer.

B Byte Modifier

The byte modifier indicates that the conversion is to take place from or into an internal field that is one-byte long. The byte modifier may be used for C, I, O, P, or X conversion. When using the B modifier with I conversion, the number is assumed positive on output and must be positive on input. For example,

BI2, BC1, BI3

See also the subsection "General Structure of Calling Sequences" for an implicit use with block addressing.

C Character Conversion

The normal form of a character format term is

Cw.n

where

"w" is the width of the external line-image, and
"n" is the width of the internal character string.

Output:

"n" characters are left-justified and placed into a output field "w" columns wide with trailing output fill characters added if necessary. The output fill character is initially a blank. If the R modifier is specified, "n" characters are right-justified and placed into the output field with preceding output fill characters added if necessary. If "w" is omitted, a default value of 1 is assumed. If "n" is omitted, "n" is set to the value of "w". If both "w" and "n" are omitted, but a decimal point appears in the format term, a default value of 1 for "w" is assumed and "n" is set to the value of "w". If standard format I/O is used, a default of 1 for "n" is assumed, and the output character string is inserted into the line-image with literal-break characters surrounding it. The maximum allowable value for "w" is 132. For example, let the argument be a seven-byte character string, "ABCDEFG"; then

```
C7.1   gives   A▪▪▪▪▪▪
C7.3   gives   ABC▪▪▪▪
RC7.3  gives   ▪▪▪▪ABC
C7.7   gives   ABCDEFG
C7     gives   ABCDEFG
C3     gives   ABC
C.1    gives   A
C.3    gives   A
C.7    gives   A
C.     gives   A
C      gives   'A'
```

▪ represents a blank.

Input:

The first "n" of "w" characters are taken from the input line-image and are placed in "n" bytes. If the R modifier is specified, the last "n" of "w" characters are used. If "w" is omitted, a default value of 1 is assumed. If "n" is omitted, "n" is set to the value of "w". If both "w" and "n" are omitted and no decimal point appears in the format term, then standard format input is assumed (see the subsection "Standard Format I/O"). For example, if columns 1 through 7 contain the character string "ABCDEFG", then

```
C1.7   gives   A▪▪▪▪▪▪
C6.7   gives   ABCDEF▪
RC6.7  gives   ▪ABCDEF
C7.10  gives   ABCDEFG▪▪▪
```

Applicable Modifiers:

B, D, DD, H, L, R, W, Z - for input or output
N - for input only
A - for output only

The default value for both "w" and "n" is 1.

D Doubleword Modifier

The doubleword modifier in a format term indicates that the conversion is to take place from or into an internal field that is eight bytes long and that begins on a doubleword boundary. However, if DD appears in a format term, conversion is to take place from or into an internal field that is sixteen bytes long and that begins on a doubleword boundary. The doubleword modifier may be used for C, E, F, I, O, P, and X conversion. For E and F conversions, when DD appears, long-precision floating-point numbers are assumed. For example,

DE, DDE1.30.40, DE0.15.22

See also the subsection "General Structure of Calling Sequences" for an implicit use with block addressing.

E Exponential Floating-Point (Real) Conversion

The normal form of an exponential floating-point format term is

Em.n.w

where

"m" is the number of digits to the left of the decimal point,
"n" is the number of digits to the right of the decimal point, and
"w" is the total external field width.

Output:

If "m", "n", and "w" are omitted and no decimal points appear in the format term, the default values of 1 and 3 for "m" and "n" for E-type formats are used; "w" is taken as m+n+6 in this case (the 6 includes the sign, the decimal point, and the four characters in the exponent). If either "m" or "n" is omitted (but either a decimal point is present or "w" is present, implying that two decimal points are present), the default values of 1 and/or 3 are assumed for "m" and/or "n", respectively (one or the other or both may be defaulted). If "w" is omitted, it is set to m+n+6. For example, the number

123.456 would print as the following with the indicated formats:

```
123.456E+00     E3.3.11
 12.346E+01     E2.3.10
  1.235E+02     E1.3.9
   .124E+03     E0.3.8
  1.235E+02     E.3.9
  1.235E+02     E
```

Input:

If neither any widths nor any decimal points appear in the format term, standard format input is assumed (see the subsection "Standard Format I/O"). Otherwise, if "m" or "n" or both are omitted, the default values of 1 and/or 3 are assumed for "m" and/or "n", respectively. If "w" is omitted, it is set to m+n+6. If there is no decimal point in the input field, the input number is scaled by 10**-n (i.e., a decimal point is assumed to be to the left of the "n"th digit counting from the right). The appearance of a decimal point in the input image overrides any specification for "m" and "n". For example, the following numbers are all converted to an internal floating-point number equal to 123.456 with respect to the formats given:

```
123.456E+00     E
12.3456E+01     E1.4
 123456E+01     E2.4
```

Applicable Modifiers:

D, DD, W - for input or output
A, J, L, R, U, Y, Z, $, @ - for output only

The default value for "m" is 1 and for "n" is 3. The mode default is W (fullword).

An exponential floating-point number is represented by

```
[sign][digits][.][digits][E|+|-|D][digits]
```

where the "E" and "D" represent single- or double-precision, respectively. Note that a string such as 1+5 is equivalent to the string 1E+05. On output, all floating-point numbers in E-format are in the following form:

```
[sign][m digits].[n digits]E[±][exponent]
```

F Nonexponential Floating-Point (Real) Conversion

The normal form of a nonexponential floating-point term is

Fm.n.w

where

"m" is the number of digits to the left of the decimal point,
"n" is the number of digits to the right of the decimal point, and
"w" is the total external field width.

Output:

If "m" or "n" is omitted, it is replaced by its respective default value of 5 or 4, respectively. If "w" is omitted and "n" is nonzero, "w" is taken as m+n+2 (the 2 includes the sign and decimal point). If "w" is omitted and "n" is zero, "w" is taken as m+n+1. If all three widths are omitted and there are no decimal points present in the format term, the number to be converted is examined with respect to the default value for "m"; if the number has an absolute value less than 10**-1 or greater than or equal to 10**(m-1), E-type standard format is used. Otherwise, "m" and "n" are set to their default values and "w" is set to m+n+2. When "n" is zero, the decimal point is still printed, but may be suppressed by using the M modifier. In this case, the default field width is m+n+1 rather than m+n+2. If the converted term is too large to fit in the output field, the output field is filled with the current overflow character (initially an asterisk).

Note: If the F is immediately followed by another F (i.e., FF), and, if the number of digits in the converted number exceeds "w", the "w" high-order characters (including the sign, if present) of the converted number are used (the last digit is <u>not</u> rounded). For example, the number 123.456 would print as the following with the indicated formats:

```
  123.456     F3.3
   123.45     F3.2
     123.     F3.0
123.4560     F
 123.456     FF3.4.7
   123.4     FF3.3.5
    ****     F2.1
```

Input:

F-type input is identical to E-type input. See the description of E above.

Applicable Modifiers:

D, DD, W - for input or output
A, J, L, M, R, U, Y, Z, $, @ - for output only

The default value for "m" is 5 and for "n" is 4. The mode default is W (fullword).

G Base Modifier

The base modifier may be used only with I-type format terms and must appear after the specification of the field width and must be separated from the main part of the format term by modifier separator ":". The number after the G is taken as the conversion base. Conversion bases from 2 through 36 inclusive may be used. The letters A through Z may be used represent the "digits" 10 through 35, respectively. For example, I10:G16 or I10:GG converts an integer field using base-16 arithmetic.

H Halfword Modifier

The halfword modifier indicates that the conversion is to take place from or into a halfword. The halfword modifier may be used with C, I, O, P, and X conversion. For example,

IH5, HC4

See also the subsection "General Structure of Calling Sequences" for an implicit use with block addressing.

I Integer Conversion

The normal form of an integer conversion format term is

Iw

where "w" is the total external field width.

Output:

If "w" is omitted, the default value of 8 for "w" for integer fields is assumed. A field of size "w", into which an integer conversion takes place, is used.

Input:

If "w" is omitted, standard format input is assumed (see "Standard Format I/O"). If "w" is present, "w" columns on the input are scanned for an integer number and are converted.

Applicable Modifiers:

B, D, G, H, W - for input or output
A, J, L, R, U, Y, Z, $ - for output only

The default value for "w" is 8. The default mode is W (fullword). Note: See the description of the byte flag B for a restriction.

J Ignore Field Width Modifier

The ignore field width modifier causes the external field width to be taken with the least number of characters which can accommodate the converted field without the use of any fill characters. The J modifier may be used with E, F, I, O, P, or X conversion. Note that with the specification of J, the R, L, and Z modifiers have no effect. This modifier is applicable to output only.

L Left-Justification Modifier

The L modifier in a C, E, F, I, O, P, or X output format term indicates that the field is to be left-justified with trailing output fill characters. For example, if "▪" stands for a blank, then on output, if a number were converted as

▪+1.234

according to the format term FY1.3.7, it will be converted as

+1.234▪

with the format term LFY1.3.7. On input, the L modifier pertains only to C conversion and causes the leftmost characters to be moved from the line-image.

M Suppress Decimal Point Modifier

The M modifier is used only in output with F format terms. It causes the suppression (omission) of the decimal point. For example, the number 123.456 would print as the following with the indicated formats:

```
   123     MF3.0
 123456    MF3.3
b123456    MF3.3.7
```

N Nulls Modifier

The nulls modifier is used only with character conversion on input. It specifies that the fill character (if needed) is to be

a hexadecimal zero X'00'. For example, if columns 1 through 7 contain the character string "ABCDEFG", then

```
NC1.7
```

would place the character A followed by six hexadecimal zeros into the 7-byte input area.

O "Own" Conversion

The normal form for the "own" conversion format term is

```
Om.n.w
```

where the "m", "n", and "w" fields are the same as the E-type and F-type format terms. The user may specify that he wants to do his own conversion. He may access the "m", "n", and "w" fields and their corresponding switches that indicate whether these fields are blank (if the field widths are zero). See "Additional Entry Points to IOH: Own Conversion" for information on setting up a program to use this type of conversion.

Applicable Modifiers:

```
B, D, DD, H, W, - for input or output
A, J, L, R, $ - for output only
```

P Packed Decimal

The normal form for the packed-decimal conversion format term is

```
Pw.n
```

where

"w" is the external field width, and
"n" is the internal field width (the number of internal packed-decimal bytes to be used).

Output:

If "w" and "n" are present, "n" bytes are unpacked and placed in "w" output columns; the "¬" modifier may be used to include the low-order digit of the last byte as a decimal digit and not as a sign. If standard format is specified, the internal representation is scanned until either a legal sign appears as a low-order digit or an illegal digit occurs in a byte (only numerics may appear in the high-order portion of a byte). If the "¬" modifier is specified, the scan stops at the last byte containing legal digits (after which there is a byte containing illegal digits). Omission of the "¬"

modifier implies that a sign must appear. A field width equivalent to one plus the number of possible digits is used for the output image field. If "n" is omitted, the same internal scan as that used for standard format is used, and "n" is determined accordingly. If "w" is omitted, a default value of 8 is assumed.

Input:

"w" digits are packed into "n" bytes with the sign in the low-order byte (see the "¬" modifier, if the sign is not desired). If standard format is specified, "n" is set to the minimum number of bytes into which the external field can be packed - leading zeros count as digits in this case. If standard format is not specified and if "w" is omitted, "n" is set to the minimum number of bytes into which the digits can be packed.

Applicable Modifiers:

B, D, H, W - for block-addressing
U, ¬ - for input and output
A, J, L, R, Y, Z, $ - for output only

The default value for "w" is 8.

Note: Decimal points may appear in an input P field (but only one per field); however, they are ignored. Thus, the user must know the scaling factor implied by the placement of the decimal point, if it appears.

Q Quit, If List Is Empty

The appearance of a Q in a format (except within a literal string) causes a switch to be tested. This switch is set only if the next parameter address is zero (i.e., if the next list element signifies the termination of format conversion). If the switch has been set, the line-image is closed by a call to the CLOSE routine and control returns to the user. If the switch is not set, the format scan continues after the Q. For example, with the output format

```
Q' NUMBER=',I5
```

the literal ' NUMBER=' will be printed only if there is a conversion argument for the I5 format term following. If the Q were not present in the format, there would be a portion of the output line with the literal ' NUMBER=' with nothing on the line after the literal.

R Right-Justification Modifier

The appearance of an R in a C, E, F, I, O, P or X output format term causes the output field to be right-justified with the leading output fill characters added, if needed. The output fill character is initially a blank. This is the normal case for numeric conversions. On input, the R modifier pertains only to C conversion and causes the rightmost (instead of leftmost) characters to be moved from the line-image.

S Space: Column Manipulator

The normal form for column spacing is

Sn

where "n" is a integer that indicates the number of columns to space. "n" may be either positive or negative. If "n" is omitted, it is assumed to be 1. For example, if the line-pointer is at column 18, S5 moves the line-pointer to column 23, whereas S-5 moves the line-pointer to column 13.

T Tab: Column Manipulator

The normal form for tabulation is

Tn

where "n" is the column number to which the line-pointer is to be moved. For example, T50 moves the line-pointer to column 50.

U Fill, If Zero

If the argument to an E, F, I, P, or X output format term is zero, the external field will be filled with output fill characters. The output fill character is initially a blank. On input, if the input field for an E, F, I, P, or X format is all blanks or null, the value is unchanged and the format is skipped. Normally, the value is set to zero.

V List Format Variable

Whenever a V is encountered in a format term, IOH assumes that the next item on the list points to a halfword integer; the contents of the halfword are substituted for the V in the format term. For example, if the next two list elements point to halfwords containing 5 and 8, respectively, then the format term

VFV.2.16

is equivalent to

5F8.2.16

W Fullword Modifier

The fullword modifier indicates that conversion is to take place from or into a fullword. This may be used for C, E, F, I, O, P, or X conversion. For example,

WI6, WE1.3.10

See the subsection "General Structure of Calling Sequences" for an implicit use with block addressing.

X Hexadecimal Conversion

The normal form of a hexadecimal conversion format term is

Xw.n

where

"w" is the external field width (the number of hexadecimal digits to be packed or unpacked), and
"n" is the number of bytes from or into which conversion is to take place.

Output:

"n" internal bytes are unpacked and placed in "w" line-image columns. If "w" is omitted, it is set to its default value of 8. If "n" is omitted, it is set to (m+1)/2, rounded down to the nearest integer.

Note: The field width is never exceeded in X-type output conversions. Thus, if internally (in two bytes) one had 0123 and if one specified X3 (which is equivalent to X3.2), the digit string 012 would be moved to the output field.

Input:

"w" digits in the input image are packed and right-justified in a field "n" bytes wide. If both "w" and "n" are missing and no decimal point appears in the format, standard format is assumed and "n" is set to the minimum number of bytes into which the field can be packed. Otherwise, if "w" is omitted, it is set to its default value of 8. If "n" is omitted, it is set to (m+1)/2, rounded down to the nearest integer. Note

that leading zeros are not ignored on input. The case of w>2n is detected as an error, terminating format processing.

Applicable Modifiers:

B, D, H, U, W - for input or output
A, J, L, R, Z - for output only

The default value for "w" is 8.

Y Forced Plus-Sign Modifier

The Y modifier is for output only. If the argument to an E, F, I, or P field is positive or zero, a plus sign is forced. For example, if the conversion term is I6, 12345 might result; if it is YI6, then +12345 would then result.

Z Zeros Modifier

The Z modifier can be used on input for C fields only. It specifies that the input fill character is to be a character zero for this conversion. On output, it can be used in an C, E, F, I, O, P, or X field to specify that the output fill character is to be a character zero for this conversion. If the output is numeric and right-justified (the default case), the zeros will be placed between the prefix characters "$", "+", or "-", if they appear, and the actual number. On nonnumeric output and numeric output without prefix characters, zeros will fill in on the right if L was specified or on the left if R was specified (or if L was not specified). Note: O-type conversion is considered as numeric output.

$ Floating Dollar-Sign Modifier

When the dollar-sign modifier is used in E, F, I, O, or P output format terms, a dollar sign is inserted in the output field immediately to the left of the first digit (or to the left of the sign, if it appears). Note: the dollar sign is illegal as a character in an input line-image for numeric fields.

@ Scale Factor Modifier

The scale factor modifier is used to specify a scale factor to be applied to E- or F-type output fields. The scale factor modifier is followed by a numeric scale factor. The appearance of the at sign must come after all field widths have been specified for the accompanying format term and must be preceded by the modifier separator ":".

If "p" is the number given after the at sign, then the scale factor is applied to an F-type number according to the formula:

External number = Internal number * (10**p)

The scaling factor is essentially applied after the conversion and causes the decimal point to be moved and/or zeros to be supplied in front of or after the converted number. For example, let the format be 2F5.5.11:@1,F5.5.11*; then the three numbers which would print as

.56789 .98765 .12345

according to 3F5.5.11 will print

5.67890 9.87650 .12345

Note that in F-type terms, application of the scale factor actually changes the number by some factor of ten by the movement of the decimal point.

In E-type formats, only the exponent is changed; the scale factor is added to the exponent. Thus, a number which would print as

.9321E-03

according to a format E0.4.10 will print as

.9321E-01

according to the format E0.4.10:@2.

¬ Packed-Decimal Sign Modifier

The not sign is used only for packed decimal conversion. For input, the not sign specifies that the low-order byte of the packed-decimal number is to contain two decimal digits rather than the normal sign and digit. On output, the low-order byte of the last byte is used as a decimal digit instead of a sign.

' Literal-Break Character

The prime is used as a literal-break character, i.e., as the left and right delimiter for a literal string. A literal string is used to provide characters in the format which are inserted into the line-image on output or are replaced by characters from the line-image on input. This is typically used for titles, labels, and other constant information. When a prime is encountered, all of the characters (including blanks) that follow the prime, up to, but not including, the next occurrence of a prime, are taken as the literal string; the length of the literal string is taken as

the number of such characters. There is one exception: to place a prime in the literal string, two consecutive primes are placed in the literal string where the single prime is desired. Such a pair of primes is interpreted as a single prime within the literal string, i.e., a literal prime, rather than as a literal-break character. Thus, the format

```
'THIS IS A LITERAL '''*
```

would print as

```
THIS IS A LITERAL '
```

whereas

```
'THIS IS A LITERAL ''*
```

is not a legal format since the second and third primes are interpreted as a single literal prime and the "*" is interpreted as the next character in the literal string rather than as the format terminator.

Note: For the convenience of 360/370-assembler language users, the double quote is also interpreted as a literal-break character and may be used as described above for the prime. Regardless of the literal-break character used, the left delimiter and the right delimiter for any single literal string must be the same character; the other literal-break character will not be recognized as a delimiter. In this context, a prime between delimiting double quotes is interpreted as a literal prime, and conversely, a double quote between delimiting primes is interpreted as a literal double quote. Thus, the format above could be written as

```
" THIS IS A LITERAL '"*
```

and in 360/370-assembler language as

```
FMT   DC  C'" THIS IS A LITERAL''"*'
```

rather than as

```
FMT   DC  C''' THIS IS A LITERAL ''''''*'
```

It is permissible to place a multiplicity before a literal string. For example,

```
3('THREE COPIES')
```

would produce:

```
THREE COPIESTHREE COPIESTHREE COPIES
```

" Literal-Break Character

The double quote is used as a literal-break character. See the description of the prime (') above.

( ) Group Indicators

A group of format terms is indicated by enclosing the group in parentheses. The left parenthesis may be preceded with a multiplicity to indicate the number of times the group is to be repeated. If the multiplicity is omitted, a multiplicity of 1 is assumed. For example,

```
3E1.5.15,2(I2,3F2.5.10),2C8*
```

is equivalent to

```
E1.5.15,E1.5.15,E1.5.15,I2,F2.5.10,F2.5.10,F2.5.10,I2,
F2.5.10,F2.5.10,F2.5.10,C8,C8*
```

Nested parentheses are allowed; there is no limit to the nesting depth. If the multiplicity is zero, the group is repeated zero times, i.e., it is ignored.

/ Line-Image Terminator (with reset)

The slash terminates the line-image with a reset. This causes a call to be made to the CLOSE routine followed by a call to the OPEN routine. Thus, it releases the current line-image and requests a new line-image. The line-pointer is reset to column 1.

| Line-Image Terminator (without reset)

The vertical stroke terminates the line-image without a reset. This causes a call to be made to the CLOSE routine without a succeeding call to the OPEN which means that the line-image area and the line-pointer are not reset, i.e., the line-image status remains as it was before the call to CLOSE.

. Field Width Separator

The period is used to separate field width specifications. This is used only with the C, E, F, O, P, and X format specifications.

* Format Terminator

The asterisk is the format terminator, the last logical character in the format. It causes a call to be made to the CLOSE routine.A

test is then made to determine whether or not the next conversion address is a zero (a zero adcon). If a zero adcon is not found, resumption of the format scan continues at the last zero-level left parenthesis, or, if this does not exist, at the beginning of the present format. The OPEN routine will be called before the format scan is resumed. If a zero adcon is found, control is returned to the user at the statement which follows the last call to IOH.

- Sign Inversion

The minus sign inverts the sign of a number (i.e., --40 is equivalent to 40).

% Current Line-Image Length Indicator

The percent sign indicates that the current line-image length is to be used in place of the percent sign. For example,

```
T%,S-1
```

sets the line-pointer at the second character from the end of the current line-image.

? Parameter List Terminator

The question mark causes IOH to return immediately to the user and ignores the rest of the present parameter list. The question mark is to be used in conjunction with the secondary entry point IOHETC. The user may return to the IOH via IOHETC and specify in the parameter list a new format address to be taken as the present format address. Any references to a previous format address will no longer exist. In particular, the placement of parentheses previously found and their associated multiplicities will be "forgotten". If the user specifies a null format address on entry to IOHETC (i.e., a zero in the format adcon), the format scan is resumed at the character immediately following the question mark.

: Modifier Separator

The colon separates the control portion of a format term from the optional modifiers that require counts (i.e., @, G, and A). It also separates these modifiers from each other if there are more than one of them. For example,

```
I5:G8 or F4.6.20:A50:@+5
```

Default Indicator

The pound sign is used to change default values in normal mode.

& Keyword Indicator

The ampersand is used to change default values in keyword mode.

< > Format Variable Indicators

The less-than sign '<' is used to begin a format variable and the greater-than sign is used to terminate a format variable. For example,

```
<FMTVAR>
```

defines the format variable FMTVAR. The format variable name is looked up in a user-defined symbol table, and its value is substituted in place of the format variable term.

= Format Variable Vector Index

The equal sign is used to specify a vector index format variable. The index preceding the equal sign must be greater than zero or omitted (in which case, an index of 1 is assumed). The index is used to index into a user-defined "format variable vector" that consists of halfword integers. The indexed entry from the vector "replaces" the format variable in the format term. For example, assume that the third and fourth entries in the vector contain a 4 and a 2, respectively; then the format term

```
3=F4=.4=.20
```

is equivalent to the format term

```
4F2.2.20
```

APPENDIX B: IOH CALLING SEQUENCES

In this appendix, the basic calling sequences for the IOH conversion subroutines are given in 360/370-assembler language format. Since the manual coding of these calling sequences is quite tedious, the general user is urged to use the macros provided for generating these calling sequences (see the subsection "IOH Macros").

General Structure of the Calling Sequence

The typical calling sequence for an IOH subroutine consists of four parts. A general example is given at the end of this appendix and will be referred to in the following discussion.

The first part is the executable code to call the subroutine; the other three parts are the parameter lists. When the call is made, general register 1 points to the beginning of the first parameter list PLIST1. The first two adcons in PLIST1 point to the OPEN and CLOSE routines (explained later in this appendix). The third adcon points to the head of the third parameter list PLIST3. If the third adcon is zero, PLIST3 is not present. The fourth adcon points to the head of the second parameter list PLIST2.

PLIST2 is made up of a variable number of adcons. This list has at its head a fullword adcon pointing to an halfword integer location that contains the total number of adcons following the head. If this halfword integer is positive, the rest of the list consists of fullword A-type adcons; if it is negative, the list is made up of S-type adcons. The first adcon following the head of PLIST2 points to the beginning of the format to be used by the IOH subroutine. The format must be present in calls to IOHIN or IOHOUT. It may be null in a call to IOHETC, in which case the adcon should be zero. If the format adcon is nonzero in a call to IOHETC, the contents of this adcon is taken as a pointer to a new format and all information about the previous format is erased. The adcons following the format adcon constitute a variable number of sublists. Each sublist also has a header that points to adcons in the sublist following the sublist header. These adcons (that follow the sublist header) point to the locations from which or to which I/O-conversion is to take place. The halfword integer to which the header of a sublist points may be positive or negative. If it is positive, each adcon in the sublist is used for only one conversion. If it is negative, the (negative) number must be even. The adcons in this kind of sublist are taken as pairs for block addresses. The first adcon of a pair contains the end address of the block. The block addresses may run either forwards or backwards in virtual memory. The incrementing (or decrementing) of a block address is determined by the format term being processed. If an explicit length modifier is given or if the format term has a default length, that length is used in computing the next block address, i.e., a length modifier of D will give a change of 8

bytes for the address, while an H will give a change of only 2 bytes. If there is no length modifier (as in a C, P, or X conversion), the internal field width is used to compute the next block address, i.e., C5.3 will give an increment of 3; P10.6, an increment of 6. I/O processing is terminated whenever one of the adcons in a sublist following a header is zero. A count of zero in the halfword integer constant to which the sublist header points causes IOH to interpret the next adcon in the list as a pointer to a new sublist header.

PLIST3 is of variable length. The first word of this list points to a halfword integer that contains the number of arguments in the list. The second adcon in the list points to a doubleword-aligned location that is the beginning of the user's symbol table. If this adcon is zero, no symbol table is present. The third adcon (if present) points to a fullword location containing either a logical I/O unit number or a FDUB-pointer. If this adcon is zero, input (or output) is done on SCARDS (or SPRINT or SPUNCH). Note that if the LUNIT address is given, but the symbol table is not, the count must be 2 and SYMTBL must be a fullword zero.

The three parameter lists may also be described using the following BNF notation.

<PLIST1>::=<pointer to OPEN> <pointer to CLOSE> <optional pointer to PLIST3> <pointer to PLIST2>

<PLIST2>::=<pointer to total number of arguments in PLIST2> <pointer to format> <list>

<PLIST3>::=<pointer to total number of arguments in PLIST3> [<pointer to SYMBTBL [<pointer to unit number>]]

<list>::=<sublist> | <list> <sublist>

<sublist>::=<pointer to number of arguments in sublist> <slist>

<slist>::=<argument> | <slist> <argument>

<argument>::=<legal 360/370-assembler expression>

Example of a complete IOH calling sequence:

```
         LA   1,PLIST1          Pointer to PLIST1
         LA   13,SAVAREA        Pointer to SAVEAREA
         L    15,=V(IOHSUB)     Address of desired IOH subroutine
         BALR 14,15             Branch to subroutine (where IOHSUB is
                                IOHIN, IOHOUT, IOHETC, etc.)

PLIST1   DC   A(OPEN)           Pointer to OPEN routine
         DC   A(CLOSE)          Pointer to CLOSE routine
         DC   A(PLIST3)         Pointer to third parameter list
         DC   A(PLIST2)         Pointer to second parameter list

PLIST2   DC   A(FULCNT)         Pointer to full list count (+12)
         DC   A(FORMAT)         Pointer to beginning of format
         DC   A(S1)             Pointer to first sublist count (+1)
         DC   A(ARG1)           Pointer to single argument
         DC   A(S2)             Pointer to second sublist count (-4)
         DC   A(ARRAY)          Beginning address of first block
         DC   A(ARRAY)+100      End address of first block
         DC   A(BLCK)           Beginning address of second block
         DC   A(BLCK-10)        End address of second block
         DC   A(S3)             Pointer to third sublist count (+3)
         DC   A(ARG3)           Pointer to single element
         DC   A(ARG4)           Pointer to single element
         DC   A(0)              End of I/O conversion

PLIST3   DC   A(FLCNT)          Pointer to full list count (+2)
         DC   A(SYMTBL)         Pointer to symbol table
         DC   A(LUNIT)          Logical I/O unit number or FDUB-pointer

FULCNT   DS   H'12'
S1       DC   H'1'
S2       DS   H'-4'
S3       DS   H'3'
FLCNT    DS   H'2'
```

Description of the OPEN and CLOSE Routines

The names OPEN and CLOSE to be used henceforth are generic names: they stand only for the concepts embodied in the following paragraphs. The user may supply his own specific OPEN and CLOSE routines for obtaining and releasing logical line-images. The OPEN and CLOSE routines are called by the IOH subroutines either to close out a line-image or to get a new line-image.

If the user is forming his own lists rather than using the macros (i.e., RDFMT, PRFMT, etc.), he must supply the adcons pointing to the OPEN and CLOSE routines whether these routines are provided by the system or by the user.

On input, the OPEN routine returns to IOH with general register 1 pointing to a two-word adcon area. The first adcon contains the address of the beginning of the input image. The second adcon points to a halfword location containing the length of the input image. When IOH calls the OPEN routine, register 1 points to a four-word adcon area, the fourth adcon of which points to an FDUB-pointer or logical I/O unit number. If this adcon is zero, the standard logical I/O unit is assumed (SCARDS for input, SERCOM, SPRINT, or SPUNCH for output). The OPEN routine should take this into account. Although it need not do so, a call to the CLOSE routine may cause the present line-image to be accepted so that a subsequent call to the OPEN routine will transmit information about a new line-image with which IOH is to work. When IOH calls the CLOSE routine, register 1 points to a four-word adcon area. The first adcon contains the address of the beginning of the line-image. The second adcon points to a halfword location containing the length of the line-image. The third adcon points to a halfword location containing the greatest excursion of the line-pointer during the conversion of the line-image. The fourth adcon points to the fullword logical I/O unit number or FDUB-pointer. If this adcon is zero, the standard logical unit is assumed (see above). The CLOSE routine may or may not use this information. Successive calls to OPEN without intervening calls to CLOSE should present IOH with the same input line-image. The return code must be given by the OPEN and CLOSE routines: 0 - successful, 4 - end-of-file.

On output, the OPEN routine returns to IOH with general register 1 pointing to a two-word adcon area. The first adcon points to the beginning of the output line-image. The second adcon points to a halfword location containing the length of the line-image. Note that IOH only inserts converted terms in the line-image; it does not blank out the line-image before processing - it is the responsibility of the OPEN routine to do this if this is desired (the normal procedure). The OPEN routines provided by the system will blank out the line-image. On a call to the CLOSE routine, register 1 points to a four-word adcon area. The adcons point respectively to the line-image, the halfword containing the length of the line-image, the halfword containing the greatest excursion of the line-pointer, and a fullword containing the logical I/O unit number or an FDUB-pointer. If the fourth adcon is zero, the standard logical unit is assumed (see above). Again, use of this information is at the discretion of the user. The return code must be given by the OPEN and CLOSE routines: 0 - successful, 4 - end-of-file.

The following examples illustrate the use of OPEN and CLOSE routines.

(1) Return from an OPEN routine to IOH

```
          LA    1,PLIST     Pickup address of return list
          SR    15,15       Return code
          BR    14          Return
           .
           .
           .
PLIST     DC    A(IMAGE)
          DC    A(AH)
           .
           .
           .
AH        DC    H'256'      Length of line-image region
IMAGE     DS    256C        Line-image region
```

(2) Call from IOH to a CLOSE routine

```
          LA    1,LIST      Pointer to parameter list
          L     15,=V(CLOSE)  Address of CLOSE routine
          BALR  14,15
           .
           .
           .
LIST      DC    A(IMAGE)    Pointer to line-image region
          DC    A(COUNT)    Pointer to length of line-image
                            region
          DC    A(LASTCOL)  Pointer to highest excursion of
                            line-pointer
          DC    A(LUNIT)    Pointer to logical I/O unit number
           .
           .
           .
IMAGE     DS    256C        Line-image region
COUNT     DS    H           Line-image length
LASTCOL   DS    H           Highest excursion of line-pointer
LUNIT     DS    F           Logical I/O unit number or
                            FDUB-pointer
```

MTS 14: 360/370 Assemblers in MTS

May 1983

May 1983

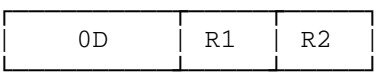
EXTENSIONS TO THE AMDAHL 470/5860 OPERATIONS

MTS running on the Amdahl 5860 computer that is installed at the University of Michigan supports four machine instructions that are not standard to the IBM System/370 architecture. Two of these instructions are executed by the Amdahl 5860 hardware and hence are as fast as other similar instructions; the other two are simulated by the supervisor in MTS and are much slower. If MTS is running on a non-Amdahl 470/5860 computer, all four instructions are simulated. The instructions are described below as they would appear in the IBM publication IBM System/370 Principles of Operation, form GA22-7000, if they were standard. This section may be considered as a local addendum.

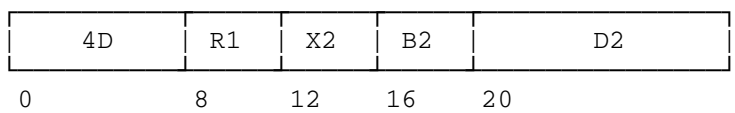
EXTENDED-BRANCH OPERATIONS

Branch and Store

BASR R1,R2 {RR}



BAS R1,D2(X2,B2) {RX}



These instructions are executed by the Amdahl 470/5860 hardware.

The 24-bit updated instruction from the current PSW is loaded as link information into the general register designated by R1; the high-order 8 bits of R1 are set to zero. Subsequently, the instruction address is replaced by the branch address.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of bit positions 8-31 of the general register designated by R2 are used as the branch address. However, when the R2 field contains zeros, the operation is performed without branching. The branch address is computed before the link information

Condition Code: The code remains unchanged.

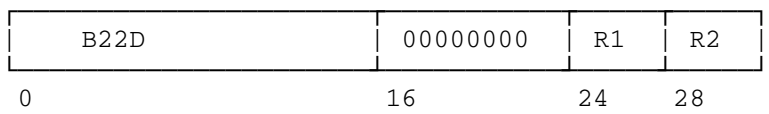
Program Interruptions: None.

Note: This instruction was designed for use with programs using 31-bit addressing. 31-bit addressing in not available in MTS at the present time.

EXTENDED-PRECISION FLOATING-POINT OPERATIONS

Divide

DXR R1,R2 {RR, Extended Operands}



This instruction is supported in MTS by simulation within the supervisor.

The extended first operand (the dividend) is divided by the extended second operand (the divisor) and replaced by the extended normalized quotient. No remainder is preserved.

Floating-point division consists of characteristic subtraction and fraction division.

The divisor is prenormalized. The dividend is prenormalized if the absolute value of the normalized dividend fraction is smaller than the absolute value of the normalized divisor fraction; otherwise, the dividend fraction is shifted so as to introduce one leading hexadecimal zero, and the characteristic is adjusted accordingly, thus yielding the adjusted dividend. The adjusted dividend has a fraction of 28 or 29 hexadecimal digits. The divisor characteristic is subtracted from the adjusted dividend characteristic, and the difference plus 64 is used as the quotient characteristic. The above process yields a normalized quotient without requiring postnormalization or right shifting of its fraction.

The quotient fraction has 28 hexadecimal digits and is developed such that it is the largest number for which the absolute value of the product of the quotient and divisor fractions is either equal to or smaller than the absolute value of the adjusted dividend fraction. All digits of the dividend and divisor fractions participate in the operation, and the dividend fraction is considered to be extended with low-order zeros.

The sign of the quotient is determined by the rules of algebra, unless the quotient is made a true zero, in which case its sign is made plus.

Unless the quotient is made a true zero, the characteristic, sign, and high-order 14 hexadecimal digits of the normalized quotient fraction (the high-order quotient) replace the high-order part of the first operand. The low-order 14 hexadecimal digits of the quotient fraction replace the low-order fraction of the first operand. The low-order sign is made equal to the high-order sign. The low-order characteristic is made 14 less than the high-order characteristic unless subtraction of 14 causes it to become less than zero, in which case it is made 128 greater than its correct value.

An exponent-overflow exception is recognized when the characteristic of the normalized quotient exceeds 127 and the fraction of the quotient is not zero. The operation is completed by making the high-order characteristic 128 less than the correct value. If the low-order characteristic also exceeds 127, it is decreased by 128. The quotient fraction and sign remain unchanged. A program interruption for exponent overflow then occurs.

An exponent-underflow exception exists when the characteristic of the normalized quotient is less than zero and neither operand fraction is zero. If the exponent-underflow mask bit is one, the operation is completed by making the characteristics of both parts 128 greater than their correct values, and a program interruption for exponent underflow occurs. The quotient fraction and its sign remain unchanged. If the exponent-underflow mask bit is zero, program interruption does not take place; instead, the operation is completed by making both the high-order and low-order parts of the quotient a true zero. Exponent underflow is not recognized when the low-order characteristic is less than zero but the high-order characteristic is zero or above. Similarly, exponent underflow is not recognized when one or both of the operands underflow during prenormalization but the quotient can be expressed without encountering exponent underflow.

A floating-point divide exception is recognized when the divisor fraction is zero. The operation is suppressed, and a program interruption for floating-point divide occurs.

When the dividend fraction is zero, the quotient is made a true zero, and a possible exponent overflow or exponent underflow is not recognized. A division of zero by zero causes the operation to be suppressed and a program interruption for floating-point divide to occur.

Resulting Condition Code:

The code remains unchanged.

Program Interruptions:

Specification: The R1 or R2 field designates a register other than 0 or 4, or bit positions 16-23 do not contain zeros. The operation is suppressed.

Exponent Overflow: The characteristic of the normalized quotient exceeds 127, and neither operand fraction is zero. The operation is completed.

Exponent Underflow: The characteristic of the normalized quotient is less than zero, neither operand fraction is zero, and the exponent-underflow mask bit is one. The operation is completed.

Floating-Point Divide: The divisor fraction is zero. The operation is suppressed.

SEARCH LIST INSTRUCTION

This instruction is supported in MTS by simulation within the supervisor. This support exists only for compatibility with the IBM System/360 Model 67; the execution time is slow and therefore is not recommended for general use.

The description of this instruction is given in the section "Extensions to the System/360 Model 67 Operations" in this volume.

May 1983

EXTENSIONS TO THE SYSTEM/360 MODEL 67 OPERATIONS

The IBM System/360 Model 67 that was installed at the University of Michigan from 1967 to 1974 had several nonstandard instructions. They are described below as they would appear in the IBM publication IBM System/360 Principles of Operation, form GA22-6821, if they were standard. This section may be considered a local addendum. Other Model 67 systems may not have these operations installed; System/360 machines that are not Model 67 systems do not have these instructions. These instructions are made available in *ASMH via the INSTSET macro instruction as described in this volume.

The IBM System/370 Model 168, the Amdahl 470, and the Amdahl 5860 computers that have subsequently replaced the Model 67 do not have these instructions[1]. These machines do have an extended-precision floating-point feature which is not precisely the same as that described below. Furthermore, the System/370 instruction set in a few cases uses the same operation codes but for different functions. For example, the extended add (ADDR) described below has an operation code of hex 26; on a 370-compatible machine, the operation code of hex 26 is used for extended multiply (MXR). Thus, object modules from the Model 67 which use the extended-precision floating-point instructions may run on a 370-compatible machine but will produce incorrect results. Use of other extended operations listed below for which there is not an operation code on the System/370 will produce an operation exception. The 370 instruction set is described in the IBM publication IBM System/370 Principles of Operation, form GA22-7000.

EXTENDED-PRECISION FLOATING-POINT OPERATIONS

The following extended-precision operations are similar to the corresponding long-precision operations with the following differences:

The address of the first operand location must be 0 or 4; otherwise a specification exception is recognized, the instruction is suppressed, and a program interrupt occurs. In the RR format, the second operand may be any one of the floating-point registers. The contents of the first operand location are replaced by a high-order result (64 bits) and the contents of the next higher addressed register, 2 or 6, are replaced with a low-order result (64 bits). It should be noted that both

---

[1]The SLT instruction is supported in MTS by simulation. This support exists only for compatibility with the Model 67; the execution time is slow and therefore is not recommended for general use.

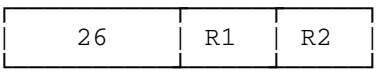
operands are long-precision (64 bits) and the result is extended-precision (128 bits). Double-precision arithmetic must still be programmed; e.g., double-precision addition can be accomplished with four ADD or ADDR operations.

The sign, characteristic, and high-order 14 hexadecimal digits of the normalized fraction, the high-order result, is placed in the first operand location. The low-order result is placed in the next higher addressed register. The low-order sign, bit 0 of this register, is made equal to the high-order sign; the low-order characteristic, bits 1-7, is made 14 less than the high-order characteristic; and the low-order fraction occupies bits 8-63.

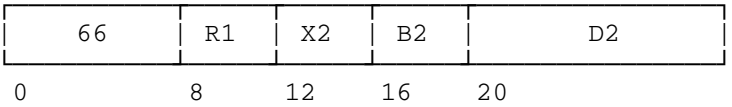
Exponent overflow occurs if the final characteristic of the high-order result exceeds 127. The operation is terminated and a program interruption occurs. Exponent underflow occurs when the characteristic of the low-order result is less than zero. The low-order result is made true zero and a program interruption occurs if the corresponding mask bit is one. The high-order result is not changed if its characteristic is in the representable range. When the characteristic of the high-order result becomes zero, both high-order and low-order results are made true zero; an exponent underflow exists and a program interruption occurs if the corresponding mask bit is one.

Add Double

ADDR R1,R2 {RR,Long Operands}



ADD R1,D2(X2,B2) {RX,Long Operands}



The second operand is added to the first operand and the normalized extended-precision sum is placed in the first operand location and the next higher addressed location.

The intermediate sum consists of 28 hexadecimal digits and a possible carry; no guard digit is retained. After addition, the intermediate sum is left-shifted as necessary to form a normalized fraction; vacated low-order digit positions are filled with zeros and the characteristic is reduced by the amount of shift.

When the intermediate sum is zero and the significance mask bit is one, a significance exception is recognized and a program interruption occurs. There is no normalization; the intermediate sum characteristic remains unchanged and becomes the high-order characteristic. The low-order characteristic is 14 less than the high-order characteristic; if it underflows, the low-order sum is made true zero and a significance exception rather than the underflow exception is recognized. In the case that the significance mask bit is zero, no significance exception is recognized and the high-order and low-order sums are made true zero. Exponent underflow cannot occur in this case.

The sign of an intermediate sum with zero fraction is always positive.

Condition Code:

0 Result fractions are zero
1 High-order fraction is less than zero
2 High-order fraction is greater than zero
3 Result exponent overflows

Program Interruptions:

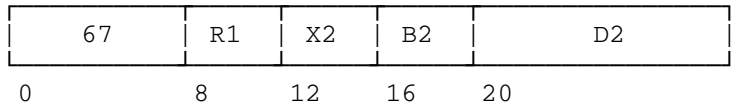
Addressing (ADD only)
Specification
Significance
Exponent overflow
Exponent underflow

Programming note: The sign of the low-order sum agrees with that of the high-order sum, except when the low-order fraction is a true zero. The low-order sum may be unnormalized.

Subtract Double

SDDR R1,R2 {RR,Long Operands}

SDD R2,D2(X2,B2) {RX,Long Operands}

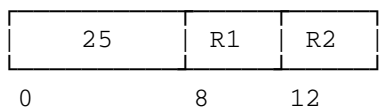


The second operand is subtracted from the first operand and the normalized extended-precision difference is placed in the first operand location and the next higher addressed location.

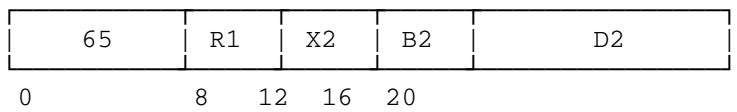
SUBTRACT DOUBLE is similar to ADD DOUBLE except that the sign of the second operand is inverted before addition.

Multiply Double

MDDR R1,R2 {RR, Long Operands}



MDD R1,D2(X2,B2) {RX, Long Operands}



The normalized extended-precision product of the second operand (multiplier) and the first operand (multiplicand) is placed in the first operand location and the next higher addressed location.

The intermediate product consists of 28 hexadecimal digits. It is left-shifted as necessary to form a normalized fraction; vacated low-order digit positions are filled with zeros and the characteristic is reduced by the amount of the shift.

When all 28 result fraction digits are zero, both high-order and low-order results are made true zero without exponent underflow and exponent overflow causing a program interruption.

There is no program interruption for lost significance.

Condition Code: The code remains unchanged

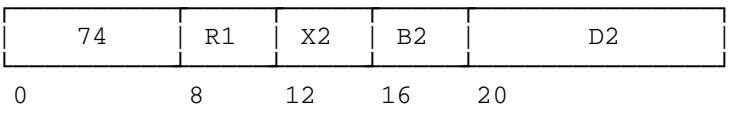
Program Interruptions:

- Addressing (MDD only)
- Specification
- Exponent overflow
- Exponent underflow

MIXED-PRECISION FLOATING-POINT OPERATIONS

Load Mixed

LX R1,D2(X2,B2) {RX, Long Operand, Short Operand}



The short-precision second operand is placed in the long-precision first operand location. The second operand is not changed. The low-order half (bits 32-63) of the result register R1 is made zero.

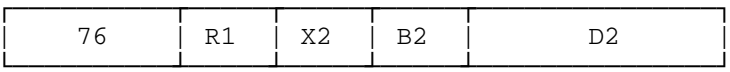
Condition Code: The code remains unchanged

Program Interruptions:

Protection
Addressing
Specification

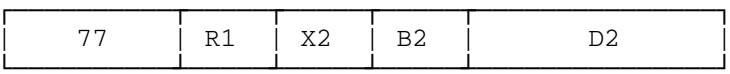
Add Mixed

AX R1,D2(X2,B2) {RX, Long Operand, Short Operand}



Subtract Mixed

SX R1,D2(X1,B2) {RX, Long Operand, Short Operand}



The short-precision second operand is added (subtracted) to the long-precision first operand, and the normalized result is placed in the first operand location. The second operand is not changed.

ADD MIXED (SUBTRACT MIXED) is similar to ADD NORMALIZED (SUBTRACT NORMALIZED) with long-precision operands, except that the second operand is short-precision and is extended to the long-precision format with low-order zeros (bits 32-63) prior to addition.

Condition Code

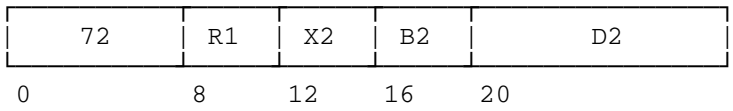
0 Result fraction is zero
1 Result is less than zero
2 Result is greater than zero
3 Result exponent overflows

Program Interruptions:

Protection
Addressing
Specification
Significance
Exponent overflow
Exponent underflow

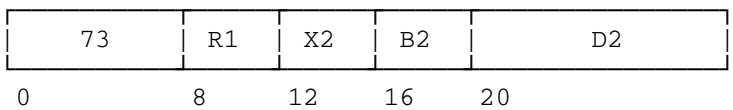
Multiply Mixed

MX R1,D2(X2,B2) {RX, Long Operand, Short Operand}



Divide Mixed

DX R1,D2(X2,B2) {RX, Long Operand, Short Operand}



For MULTIPLY MIXED, the normalized product of the short-precision second operand (multiplier) and the long-precision first operand (multiplicand) is placed in the first operand location. For DIVIDE MIXED, the long-precision first operand (dividend) is divided by the short-precision second operand (divisor) and the long-precision quotient is placed in the first operand location. The second operand is not changed.

MULTIPLY MIXED (DIVIDE MIXED) is similar to MULTIPLY (DIVIDE) with long-precision operands, except that the second operand is short-precision and is extended to the long-precision format with low-order zeros (bits 32-63) prior to the operation.

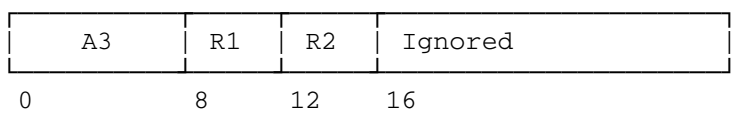
Condition Code: The code remains unchanged

Program Interruptions:

- Protection
- Addressing
- Specification
- Exponent overflow
- Floating-point divide (DX only)

SWAP REGISTER INSTRUCTION

SWPR R1,R2 {RS, Long Operands}



The first operand replaces the second operand and the second operand replaces the first. The first operand is a doubleword contained in an adjacent pair of general registers; R1 designates the leftmost register of the pair. The second operand is a doubleword contained in the floating-point register R2, R1 must be even and R2 must 0,2,4, or 6; otherwise a specification exception is recognized.

Bit positions 16-31 of this instruction are ignored.

Condition Code: The code remains unchanged

Program Interruptions:

- Specification

SEARCH LIST INSTRUCTION

The instruction SEARCH LIST is designed to facilitate searching a chained list of data entries (elements) to locate elements by comparison with a data field value, key value, and/or entry count. The instruction is designed for searching list-organized data where the relative positions of the data field, key field, and chain addresses within an element are constant for the full set of elements traversed in a single instruction execution.

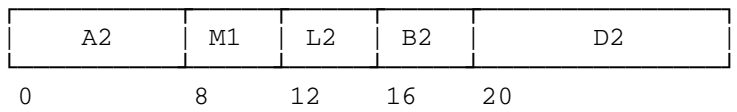
The instruction makes use of up to six implied general registers (0 through 5) to hold the dynamic and fixed parameters of the search and one explicitly referenced general register to provide a base for the origin of the list-structured data area being searched.

The instruction is designed so that legitimate relocation exceptions, which may occur during the course of a chained search in a dynamic paging environment, will leave the general registers in a stage where the search can be resumed by supervisor action. Thus the effects of dynamic paging can be transparent to the using program.

Name	Mnemonic	Type	Exceptions	Code
SEARCH LIST	SLT	RS	P,A,S,Reloc.	A2

SLT M1,L2,D2(B2) OS format

SLT M1,D2(L2,B2) TSS format



A sequence of list elements is searched by a data comparison and a key test.

The list is located in storage relative to the base address in the general register designated by B2. Each list element is identified by an element address. Within a list element there is found a chain address, a data field and a key. The chain addresses establish the sequence of the list elements. The data and keys are used as criteria for completing the search. The types of tests to be performed are specified by the M1 field of the instruction. The comparand for the data, the mask for the key and the necessary offsets to locate the data field and the key within the list element reside in the general registers. The general registers also contain a count which limits the number of elements searched. When the search ends, the address of the last list element inspected, the current element address, as well as the address of the preceding list element, the predecessor address, are inserted in general registers.

Element Addressing

As searching proceeds from element to element, the address of the list element to be inspected, the current element address, is obtained by adding the chain address of the preceding element and the base address designated by B2. The current element address is inserted in general register 2 when the necessary data, key and count tests of the current element are completed and remains available in that register during the testing of the subsequent list element. As the current

element address is inserted in general register 2, the former content ofthis register is inserted as a predecessor address in general register 1.

The chain address occupies the word designated by the sum of the element address in register 2 and the displacement D2.

When searching is initiated, the contents of general register 2 are used to locate a chain address which points to the first element. Thus the first element address is the sum of the base address designated by B2 and the chain address at the location designated by the contents of register 2 and D2.

An element must be located on a doubleword boundary, i.e., bits 29-31 of the element address must be zero, otherwise a specification exception is recognized. The chain address must be located on a fullword boundary, i.e., bits 30-31 of the D2 field must be zero, otherwise a specification exception is recognized. In either case, a program interruption results and the operation is terminated.

A B2 field of zero indicates the absence of the base address and the use of the chain address as an element address.

In the 24-bit mode, the chain address occupies bits 8-31 of its word location. Bits 0-7 of this word location are ignored. As the element address is inserted in general register 2, the high-order eight bits are made zero. The high-order 8 bits of the offsets in general registers 4 and 5 are ignored. The entire 32-bit contents of general register 2 are placed in general register 1 when the predecessor address is recorded. In the 32-bit mode, all bits of word locations and registers participate.

Data Comparison

The contents of the data field within the list element are compared with the contents of general register 3. The criterion for comparison is determined by bits 8-10 of the mask field M1 of the instruction.

The data field is variable in length and is located at the byte address designated by the sum of element address and the contents of general register 4. The length of the data field is designated by the length field L2, bits 12-15 of the instruction. The length specification must be 0, 1, 2, or 3, designating a length of 1, 2, 3, or 4 bytes, respectively. Otherwise, a specification exception is recognized, a program interruption is taken and the instruction is suppressed.

The comparand in general register 3 is right aligned in the register. Only the number of bytes specified by L2 is used in the comparison. Any other bytes in the register are ignored. Comparison is logical and temporarily sets the condition code as in the instruction Compare Logical. Bits 8, 9, and 10 of the mask field M1 identify a successful condition as in the instruction Branch on Condition. When a comparison

condition has the corresponding bit in the mask field M1 set to one, the search is completed, and the condition code is set to indicate that the search is completed by a successful comparison.

When all three mask bits, 8, 9, 10, are zero, no comparison is identified as successful. In this case no comparison is made and the contents of general registers 3 and 4 are not used.

Key Test

The key within the list elements is tested by a key mask in general register 0. The key occupies an 8-bit byte and is located at the byte address designated by the sum of the element address and the contents of general register 5. The key mask occupies bits 16-23 of general register 0.

The key test is performed as in the instruction Test under Mask. The criterion for a successful test is determined by bit 11 of the mask field M1 of the instruction. For any one bit in the key mask, the corresponding bit in the key is tested for a one. When bit 11 is zero and any of the bits so tested is one, or when bit 11 is one and all the bits so tested are one, the key test is considered successful. In that case the search is completed and the condition code is set to indicate that the search is completed by a successful key test.

When all bits of the key mask are zero, no key test is performed and the contents of general register 5 as well as bits 16-23 of general register 0 are not used.

Counting

The count, which specifies the number of elements to be inspected, resides in general register 0, bits 24-31. A count of zero specifies 256 elements to be inspected, the maximum number of elements which can be inspected in one instruction.

After the data comparison and key test are completed, the count is reduced by one.

When no successful comparison or key test has occurred, the new count is tested for zero. If zero, the search is completed and the condition code is set to indicate that the search is completed due to count run-out. If the count is not zero, the search continues.

Ending

When searching is completed due to a successful comparison or key test or exhausted count, the current element address resides in register 2, the predecessor address resides in register 1, and the count

indicates the number of elements not inspected. The chain address of the current element is not used in this case.

The contents of general register 0 bits 0-15 are unpredictable at the end of the instruction.

This instruction differs from others in that it may be terminated by a relocation exception rather than being suppressed. The relocation exception may be due to the chain address location, the data field location, or the key location. The condition code is unpredictable. In each case register 2 contains the address of the element which precedes the element tested. Thus the operation may be resumed by reissuing the instruction.

Condition Code:

0 Unsuccessful comparison or key test, completion due to count
1 Successful comparison, unsuccessful key test
2 Unsuccessful comparison, successful key test
3 Successful comparison and key test

Program Interruptions:

Operation (if feature is not installed)
Addressing
Specification
Protection
Relocation

Programming Notes

Upon instruction completion, condition code zero indicates that all the elements specified by the initial count were inspected without encountering a successful data comparison or key test. However, due to a successful comparison or key test a nonzero condition code may also occur for the last element specified by the initial count.

The recording of the predecessor address in general register 1 facilitates insertion or deletion of list elements. When searching is completed with the first element, the contents of register 1 point to a virtual predecessor element which contains the first chain address. Thus uniform insert and delete procedures can be used.

When the B2 field specifies general register 2, the chain address designates the location of the successor element relative to the current element location. When the B2 field specifies general register 1, the location of the successor element is designated relative to the predecessor element.

Warning: Because IBM has improperly engineered this instruction, B2 cannot specify general registers 1, 2, or 4.

Warning: Because IBM has improperly engineered this instruction, a program interrupt will occur under the following conditions:

(1) Instruction bits 8-10 are not all zero, indicating that the data comparison feature is being used,
(2) The count in GR0 is reduced to zero without a data comparison succeeding, and
(3) The next element address computed from the chain address field of the last element and the register specified in the B2 field of the instruction is not a valid address.

This interrupt does not occur when only the key test is performed or when the data comparison succeeds on the last element.

It is almost certain that the above two errors will never be fixed.

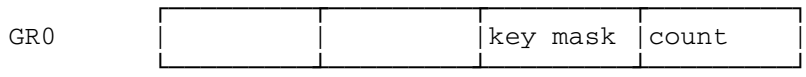
Engineering Notes

Performance improvements are obtained for each of the following cases:

(1) Instruction bits 8, 9, and 10 are zero, indicating the absence of a data comparison.
(2) Bits 16-23 of general register 0 are zero, indicating the absence of a key test.
(3) The key and chain address are located in the same doubleword.

Summary

Registers:



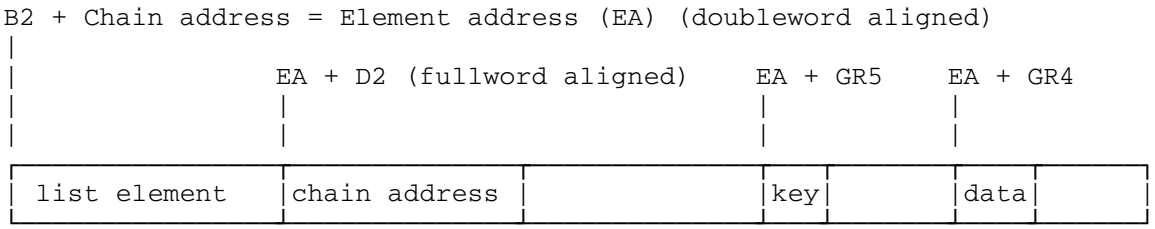
GR1 Previous element address (doubleword aligned)

GR2 Current element address (doubleword aligned)

GR3 Data for comparison (right-justified)
Contents(GR3) - Data in element

GR4 Offset in element for data

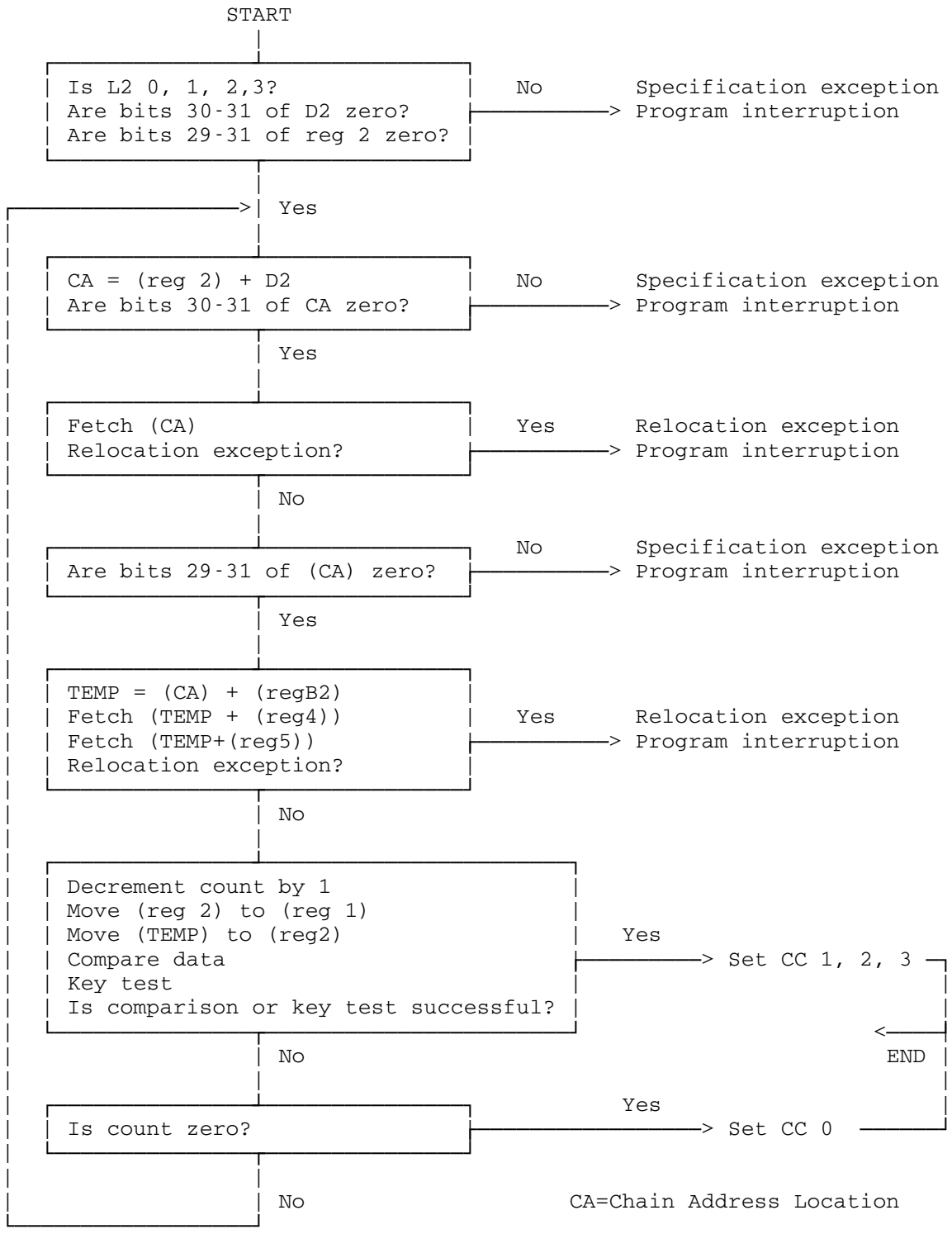
GR5 Offset in element for key



Initializing:

```
L     B2,=A(BASE)
LA    GR2,=A(firstlistelement-BASE)
SLT   M1,L2,D2(B2)
BZ    *-4  (If infinite count desired)
...
```

SEARCH LIST Instruction Flow



May 1983

THE MACRO-LIBRARY EDITOR

The MTS macro-library editor is used for the editing of macro libraries containing 370 assembler language macros. Many OBJUTIL facilities, such as replacing macros, are incorporated in this macro-library editor. The macro-library editor provides facilities to replace, add, delete, or correct a macro. The macro-library editor is available in the file *MACUTIL and is invoked via the $RUN command. The macro-library editor uses the following MTS logical I/O units:

SCARDS - the input file containing the macros to be replaced.
SPRINT - printed output produced by the macro-library editor.
SERCOM - diagnostic messages.
GUSER - a sequence of commands or user responses in conversational mode.
0 - default unit for the macro library to be edited.

Those users who want only to update their macro library can simply issue the following command:

```
$RUN *MACUTIL SCARDS=inFDname 0=editFDname
```

In the above case, macros are read from "inFDname", and written on "editFDname" replacing any previous macros. The logical structure of the macros is completely preserved except that the duplicate macros from "inFDname" are discarded. Unless the parameter REPLACE is specified, additional macros are also added to the macro library.

A typical example is a file consisting of several macros in the file MAC. While assembling the macros, the user discovers an error in one of the macros. This error may be corrected in the source file by the MTS file editor. After the macro is checked, the macro-library editor may be used to replace the erroneous macro in the macro library as follows:

```
$RUN *MACUTIL SCARDS=MAC 0=MACLIB
```

In this example, the source code for the macros replaces corresponding macros in the file MACLIB. In addition, any macro that exists in MAC but not in MACLIB is also added.

Users who only want to build the directory in the macro library can simply issue the following command:

```
$RUN *MACUTIL 0=editFDname PAR=BUILDIR
```

The first record of the file "editFDname" should not start at line 1; otherwise, MACUTIL will not be able to build the directory properly (the

directory starts at line 1). It is recommended that the first record start at line 1000; this will allow enough room for the directory.

The PAR field allows the user some control over the processing done by the macro-library editor. Those users who wish to use the more advanced features of the macro-library editor must use the macro-library editor command language described below. Commands are read from GUSER and printed output is written on SPRINT. The typical $RUN command to use the macro-library editor in command mode is:

```
$RUN *MACUTIL
```

Command input is terminated by an end-of-file or by the STOP command.

The following parameters may be specified in the PAR field of the $RUN command. The parameters must be separated by commas or blanks. Parameters may be negated by "-", "¬", "NO", or "N". The minimum acceptable abbreviation for each parameter is underlined.

BREAK=n

Each new macro or copy section added to the edit file begins at the next highest multiple of the line number "n" after the last line in the macro library. Lines currently in the edit file are not changed to reflect the new value. The default value is 100.000. No fractions are permitted.

BUILDIR

The BUILDIR parameter specifies that the macro library editor is to produce a directory in the macro library. The library is assumed to have no directory but just a series of macros and/or copy sections. The default is NOBUILDIR; i.e., the macro library should not have the directory generated.

COMSAVE

The COMSAVE parameter specifies that the macro-library editor is to retain comments before the MACRO header and all macro comments starting with ".*". NOCOMSAVE specifies that such comments should be deleted. Assembler-G does not allow ".*" comments before MACRO headers. The default is COMSAVE.

EMPTY

The EMPTY parameter specifies that the macro library assigned to unit 0 is to be emptied before use. NOEMPTY specifies that the macro library is not to be emptied. The default is NOEMPTY. EMPTY has no effect in command mode.

FULL

The FULL parameter specifies that macros used in the ADD, COPY, INCLUDE, LIST, REPLACE, and UPDATE commands are to be fully listed on SPRINT. The default is NOFULL.

HDRGEN

The HDRGEN parameter specifies that MACUTIL generate a copy-section header whenever a copy section is added with the NAME modifier in the ADD, INCLUDE, REPLACE, and UPDATE commands. NOHDRGEN specifies that the copy-section header should not be generated. The default is HDRGEN.

INCREMENT=n

The INCREMENT parameter may be used to increment each successive line in the macro library. The default is 1.000.

LIST

The LIST parameter specifies that the macro-library editor lists each macro name, its beginning and ending line numbers, and its line count for the following commands: ADD, COPY, INCLUDE, REPLACE, and UPDATE. The default is NOLIST.

QUIT

The QUIT parameter specifies that if the macro-library editor encounters any errors in batch mode, the user is signed off. NOQUIT specifies that the batch user is not to be signed off if an error occurs. The default is NOQUIT.

SEQ

The sequence numbers in columns 73-80 are normally not removed. But if SEQ=OFF, they are deleted and the SEQ parameter can be used to truncate any new macros. The default is SEQ=ON.

SORT={LNR|NAME}

The SORT parameter controls how the LIST command will list macros. If SORT=NAME is specified, the macro-library editor will list macros alphabetically. If SORT=LNR is specified, macros are listed according to their beginning line numbers in ascending order. The default is SORT=NAME.

START=n

The START parameter specifies that, if a macro library is empty or is to be renumbered, the first line of a first macro should start at MTS line number n. The parameter should be an integral number, and the default is 1000.

TERSE/VERBOSE

The TERSE/VERBOSE parameters control the amount of information produced by the verification of some commands. TERSE specifies that minimal information is requested; VERBOSE specifies that full information is desired. TERSE is an antonym for VERBOSE. The default is the setting of the MTS TERSE option. This parameter pair has no effect if NOVERIFY is specified.

UPDATE/REPLACE

The UPDATE/REPLACE parameters control whether the file assigned to SCARDS is an update or replacement file in no-command mode. UPDATE specifies that macros or copy sections read from SCARDS not in the edit file will be included. REPLACE specifies that new macros will be excluded. The default is UPDATE. The parameters are effective only if SCARDS is specified in the $RUN command.

VERIFY

The VERIFY parameter specifies that verification for each command is requested. NOVERIFY suppresses the verification. The default is VERIFY.

The complete description of the macro-library editor command language is given on the following pages.

MACRO-LIBRARY EDITOR COMMAND LANGUAGE

The general form of a command is:

```
commandname[@modifier]...[ operand]...
```

Modifiers may be prefixed by "¬", "-", "NO", or "N" if they are to be negated. In some commands, "slist" as an operand of a command stands for:

```
{ALLBUT|[ONLY]} symbol [[,]symbol] ...
```

The following notation conventions are used in the description of the macro-library editor command language:

...	denotes zero or more repetitions of the preceding words.
|	denotes a choice of options, e.g., x|y means choose "x" or "y".
[ ]	denotes optional words.
{ }	denotes alternatives.
___	denotes a minimum acceptable abbreviation for a word, e.g., INCLUDE indicates I is the minimum acceptable abbreviation for INCLUDE.

The following rules apply to command usage:

(1) A command starts with the first nonblank character, which need not start at position 1.
(2) There should be no embedded blanks in the command name and command modifiers.
(3) At least one blank should separate the command name and the first operand.
(4) There must be a blank or a comma between any two operands.
(5) If the last character of the current input line is a minus sign "-", the next input line will be taken as a continuation of the current line. The first character of the next line replaces the continuation character. There is no limit to the number of continuation lines, however, the total number of characters in a command line may not exceed 256. For batch use, the continuation character must be punched in column 80, since all 80 columns of a card are read.
(6) Any command or command modifier may be abbreviated by entering only an initial substring, which is underscored in each command or command modifier description.
(7) A command line beginning with an asterisk "*" is considered to be a comment and is not processed, other than possibly being echoed by the macro-library editor.
(8) A command line beginning with a dollar sign "$" is assumed to be an MTS command and is executed by a call to the system CMD subroutine.

The following table summarizes the macro-library editor commands and their applicable modifiers.

Command	Operand	Applicable Modifiers
ADD	[FROM] FDname [slist]	BREAK=, COMSAVE, FULL, HDRGEN, INCREMENT=, LIST, NAME=, SEQ, TERSE, VERBOSE, VERIFY
BUILDIR	[filename]	TERSE, VERBOSE, VERIFY
CLEAR	None	VERIFY
COMMENT	comment	None
COPY	[FROM] inFDname [TO] outFDname [slist]	BREAK=, COMSAVE, FULL, HDRGEN, INCREMENT=, LIST, NAME=, SEQ, TERSE, VERBOSE, VERIFY
CREATE	filename [lhs=rhs]...	VERIFY
DELETE	slist	TERSE, VERBOSE, VERIFY
DISPLAY	item ...	None
EDIT	filename	EMPTY
EMPTY	filename	VERIFY
EXPLAIN	[ON FDname] [item]...	EMPTY
HELP	None	None
INCLUDE	[FROM] FDname [slist]	BREAK=, COMSAVE, FULL, HDRGEN, INCREMENT=, LIST, NAME=, SEQ, TERSE, VERBOSE, VERIFY
LIST	[ON FDname] [slist]	EMPTY, FULL, SORT=
MCMD	MTS command	None
MTS	[optional command]	None
PUNCH	[ON] FDname [slist]	COMSAVE, EMPTY, SEQ, TERSE, VERBOSE, VERIFY
RENAME	old1[=]new1[[,]old2[=]new2]]...	VERIFY
RENUMBER	[name] [[start,]increment]]	BREAK=, INCREMENT=, START=, VERIFY
REPLACE	[FROM] FDname [slist]	BREAK=, COMSAVE, FULL, HDRGEN, INCREMENT=, LIST, NAME=, SEQ, TERSE, VERBOSE, VERIFY
RETURN	None	None
SET	lhs=rhs [[,] lhs=rhs]...	None
STOP	None	None
UPDATE	[FROM] FDname [slist]	BREAK=, COMSAVE, FULL, HDRGEN, INCREMENT=, LIST, NAME=, SEQ, TERSE, VERBOSE, VERIFY

May 1983

Macro-Library Editor Commands

Command: ADD [FROM] FDname [slist]

Modifiers: BREAK, COMSAVE, FULL, HDRGEN, INCREMENT, LIST, NAME, SEQ, TERSE, VERBOSE, VERIFY

Example: ADD FILE1+FILE2(100,199) ALLBUT QQ

Explanation: Macros are added to the macro library from the specified file or devices. Input is terminated by an end-of-file.

Command: BUILDIR [filename]

Example: BUILDIR MAC

Modifiers: TERSE, VERBOSE, VERIFY

Explanation: The MACUTIL reads the file, if specified, and builds the directory establishing it as a macro library. If the directory already exists, MACUTIL will ask the user if the directory is to be rebuilt. If so, the directory will be deleted and built up. If not, MACUTIL establishes the file as a macro library by simply reading in the directory.

Command: CLEAR

Modifiers: VERIFY

Example: CLEAR@NV

Explanation: The CLEAR command completely clears out the current internal representation of the macro library. The macro-library editor will then rebuild its internal representation by reading the macro library. This command is necessary whenever the user changes the macro library without letting the macro-library editor know (via MTS, MCMD, or $ commands; in particular, via an MTS $RENUMBER command).

Command: COMMENT comment

Modifiers: None

Example: COMMENT - Now we delete some macros.

Explanation: The COMMENT command is useful for documenting sequences of commands given to the macro-library editor. Note that command lines beginning with an asterisk "*" are also treated as comments.

Command: COPY [FROM] inFDname [TO] outFDname [slist]

Modifiers: BREAK, COMSAVE, FULL, HDRGEN, INCREMENT, LIST, NAME, SEQ, TERSE, VERBOSE, VERIFY

Example: COPY FROM NEWMACS TO MACLIB

Explanation: This COPY command copies macros and copy sections from an input file (possibly a macro library) to an output macro library file, which then becomes the active macro library. This command is identical to running *MACUTIL in no-command mode with SCARDS and 0 assigned; and is provided only to make that mode of operation available at the command level.

Command: CREATE filename [lhs=rhs]...

Modifiers: VERIFY

Example: CREATE MAC SIZE=100P

Explanation: The CREATE command creates a macro library. The filename is acquired and becomes the active macro library. The optional keywords are:

```
SIZE=nP
MAXSIZE=nP
TYPE=LINE
VOLUME=volname
```

The keywords are the same as those for the MTS command $CREATE except that the SIZE and MAXSIZE are expressed in terms of pages and that sequential files are not allowed.

Command: DELETE slist

Modifiers: TERSE, VERBOSE, VERIFY

Example: DELETE MACRA, MACRB

Explanation: Through the DELETE command, the macros implied by the symbols in "slist" are deleted from the macro library. This allows the replacement of macro definitions.

Command: DISPLAY item ...

Modifiers: None

Example: DISPLAY DIRECTORY

Explanation: The DISPLAY command can be used to display the current status of the macro library, its directory, or any of options that can be used by the SET command. "item" may be one of the following:

> BREAK, COMSAVE, DIRECTORY, ECHO, FULL, HDRGEN, INCREMENT, LIST, OPTIONS, QUIT, SEQ, SORT, START, STATUS, TERSE, TYPE, VERBOSE, VERIFY.

DIRECTORY displays all names of macros and copy sections as listed in the directory.

OPTIONS displays all options that can be used by the SET command.

STATUS displays the name of the macro library, the count of macros and copy sections, and the count of the lines.

Command: EDIT filename

Modifiers: EMPTY

Example: EDIT MACLIB

Explanation: The current macro library is set to filename. This command must be specified if unit 0 is not initially assigned. The macro-library editor reads the file "filename" and builds an internal directory of the location of each macro or copy section in the file. Currently, "filename" must be a single file with no explicit concatenation or explicit line-number increment specified.

The EMPTY modifier may be specified to request the macro-library editor to empty "filename" before setting it to the current macro library.

Command: EMPTY filename

Modifiers: VERIFY

Example: EMPTY MACLIB

Explanation: The specified filename is emptied and then becomes the current macro library.

Note: This command is the same as the EDIT command with the @EMPTY modifier specified.

Command: EXPLAIN [ON FDname] [item]...

Modifiers: EMPTY

Example: EXPLAIN @HDRGEN, COMMANDS

Explanation: The EXPLAIN command prints an explanation of the specified items on SPRINT or on FDname if specified. The following items can be explained:

> $, *, ?, -, ADD, BUILDIR, CLEAR, commands, COMMENT, continuations, COPY, CREATE, DELETE, description, DISPLAY, EDIT, EMPTY, everything, example, EXPLAIN, HELP, INCLUDE, LIST, macutil, MCMD, modifiers, MTS, operations, parameters, PUNCH, QUIT, RENAME, RENUMBER, REPLACE, RETURN, SET, slist, STOP, syntax, TYPE, UPDATE, @BREAK, @COMSAVE, @EMPTY, @FULL, @HDRGEN, @INCREMENT, @LIST, @NAME, @SEQ, @SORT, @START, @TERSE, @VERBOSE, @VERIFY.

Items listed in lowercase are generic and produce a general explanation; items listed in uppercase refer to a particular command, modifier, or parameter and produce a specific explanation. The EMPTY modifier may be specified to empty the output file first.

Command: HELP

Modifiers: None

Example: HELP

Explanation: The HELP command prints a list of all valid macro-library editor commands.

Command: INCLUDE [FROM] FDname [slist]

Modifiers: BREAK, COMSAVE, FULL, HDRGEN, INCREMENT, LIST, NAME, SEQ, TERSE, VERBOSE, VERIFY

Example: INCLUDE FILE1+FILE2(100,199) ALLBUT QQ

Explanation: Macros are added to the macro library from the specified file or devices. Input is terminated by an end-of-file.

Note: This is a synonym of the ADD command.

Command: LIST [ON FDname] [slist]

Modifiers: EMPTY, FULL, SORT

Example: LIST ON *PRINT*

Explanation: The LIST command allows the user to obtain information about the macros currently in the macro library. The information about the macros shows their starting and ending line numbers, and gives a count of the lines present. If the modifier @FULL is specified, then macros are listed in full. Macros are listed in alphabetic order according to their names. SORT=LNR may be specified to list the macros according to their beginning line numbers.

If "slist" is not specified, the "short" information is listed for all relevant macros. LIST information is written to SPRINT unless "ON FDname" is given, in which case it is written to the specified file or device. The EMPTY modifier may be specified to empty the output file before printing the list.

Command: MCMD MTS command

Modifiers: None

Example: MCMD EMPTY MACLIB OK

Explanation: The MTS command specified is executed by MTS, and control is returned to the macro-library editor. Alternatively, the user may issue the MTS command directly in macro-library editor command mode by prefixing it with a dollar sign, e.g., $EMPTY -MAC.

Note: The macro-library editor will not know if the user changes the active macro library.

Command: MTS [MTS command]

Modifiers: None

Example: MTS EMPTY MAC OK
MTS

Explanation: If the MTS command is specified, it is executed by MTS, and the macro-library editor may be reentered with an MTS $RESTART command. Alternatively, the user may issue just the MTS command. Control then reverts to MTS command mode in such a way that the macro-library editor may be reentered with a $RESTART command.

Note: The macro-library editor will not know if the user changes the current macro library.

Command: PUNCH [ON] FDname [slist]

Modifiers: COMSAVE, EMPTY, SEQ, TERSE, VERBOSE, VERIFY

Example: PUNCH ON -X ONLY SORT

Explanation: If no "slist" is given, all macros of the current macro library are punched on FDname. Otherwise, the specified macros are punched. In addition, if the EMPTY modifier is specified, the FDname is first emptied before punching.

Command: RENAME old1[=]new1 [[,]old2[=]new2] ...

Modifiers: VERIFY

Example: RENAME MIN=MINER

Explanation: The RENAME command causes one or more specified macros to be renamed. Renaming is done first in the directory. The macro prototype is also changed if it was the same as the old name.

Command: RENUMBER [name] [{increment|start,increment}]

Modifiers: BREAK, INCREMENT, START, VERIFY

Example:
```
RENUMBER MINE 2
RENUMBER@BREAK=1000 .5
```

Explanation: The RENUMBER command is used either to renumber a specified macro "name" or to renumber the entire macro library.

If "name" is specified in the command, only that macro will be renumbered according to "start" and "increment". The defaults are its actual beginning line number and the INCREMENT parameter, respectively. Renumbering a macro with a different first line number will not be done if the macro would have to be moved out of the usual order of the macros in the library.

If "name" is not specified, the entire macro library will be renumbered according to the BREAK, INCREMENT, and START parameters. The defaults are 100., 1., and 1000., respectively. Optionally, "start" and "increment" can be specified as operands of the command instead of corresponding modifiers. At first, lines are internally renumbered just to check if any line will not be out of range (over 99999.999). All macros are then actually renumbered starting at the specified START parameter (defaults to 1000.) with the increment=0.001. The BREAK parameter is ignored during this intermediate renumbering. Beginning with the last macro and ending with the first macro, MACUTIL renumbers each macro according to the parameters BREAK and INCREMENT. Finally, the directory of the macro library is updated with correct beginning line numbers of macros.

Command: REPLACE [FROM] FDname [slist]

Modifiers: BREAK, COMSAVE, FULL, HDRGEN, INCREMENT, LIST, NAME, SEQ, TERSE, VERBOSE, VERIFY

Example: REPLACE FROM -LOAD

Explanation: The REPLACE command reads potential replacement macros from "FDname" and selectively replaces those modules in the macro library according to "slist". Any additional macros in "FDname" are ignored. The REPLACE command can be thought of as a convenient way of performing the following operations:

```
DELETE slist
INCLUDE FROM FDname slist
```

with the additional feature that the original ordering of the macros in the macro library are preserved if possible. Note that the UPDATE command performs a very similar function.

Command: RETURN

Modifiers: None

Example: RETURN

Explanation: Control reverts to MTS command mode in such a way that the macro-library editor may be reentered via the $RESTART command. The RETURN command is identical to the MTS command with no operands specified.

Command: SET lhs=rhs[[,]lhs=rhs]...

Modifiers: None

Example: SET ECHO=ON VERIFY=OFF

Explanation: Most of the items which can be specified in a SET command are also available as modifiers to the individual commands. The SET command simply changes the global default value for such modifiers so that the same modifier values need not be given repeatedly. The available keywords are:

BREAK=n	Defaults to 100.000
COMSAVE={ON|OFF}	Defaults to ON
ECHO={ON|OFF}	(see below)

FULL={ON|OFF}	Defaults to OFF
HDRGEN={ON|OFF}	Defaults to ON
INCREMENT=n	Defaults to 1.000
LIST={ON|OFF}	Defaults to OFF
MODCHAR=character	Defaults to @
QUIT={ON|OFF}	(see below)
SEQ={ON|OFF}	Defaults to ON
SORT={LNR|NAME}	Defaults to NAME
START=n	Defaults to 1000.000 if file is empty, where the first source line will start.
TERSE={ON|OFF}	Defaults to OFF
VERBOSE={ON|OFF}	Defaults to ON (antonym of TERSE)
VERIFY={ON|OFF}	Defaults to ON

Note that most of these items may be specified as execution parameters in the PAR field of the $RUN command (see the description of the parameters available at the beginning of this section).

If ECHO is turned ON, macro-library editor commands are ECHOed on SPRINT. ECHO defaults to ON unless the commands are being entered directly from a terminal.

If the macro-library editor encounters any errors when QUIT is turned ON in batch mode, the user is signed off. QUIT defaults OFF for batch and is always OFF for conversational use.

Command: STOP

Modifiers: None

Example: STOP

Explanation: The macro-library editor terminates processing. An end-of-file in the command stream also terminates the processing.

Command: UPDATE [FROM] FDname [slist]

Modifiers: BREAK, COMSAVE, FULL, HDRGEN, INCREMENT, LIST, NAME, SEQ, TERSE, VERBOSE, VERIFY

Example: UPDATE FROM -SECTION

Explanation: The UPDATE command reads potential replacement macros from "FDname" and selectively replaces those macros in the macro library according to "slist". Any additional macros in "FDname" are also included. The UPDATE command can be thought of as a convenient way of performing the following operations:

```
DELETE slist
INCLUDE FROM FDname
```

with the additional feature that the original ordering of the macros in the macro library are preserved if possible. Note that the REPLACE command performs a very similar function.

Command Modifiers

The modifiers are prefixed by "@" or MODCHAR as set by the SET MODCHAR command (see the SET command) and appended to the commands. A modifier may be negated by prefixing it with "¬", "-", "NO", or "N".

Modifier: BREAK=n

Example: ADD@BREAK=100 MACFILE

Explanation: The BREAK modifier may be appended to the ADD, COPY, INCLUDE, RENUMBER, REPLACE, and UPDATE commands to set the beginning line number of each new macro added to the macro library to the next highest multiple of "n". The default is 100.000.

Modifier: COMSAVE

Example: ADD@¬COMSAVE ASMTMAC

Explanation: The COMSAVE modifier may be appended to the ADD, COPY, INCLUDE, PUNCH, REPLACE, and UPDATE commands. The COMSAVE modifier specifies that comments before MACRO headers and comments starting with ".*" are to be saved. The default is COMSAVE.

Modifier: EMPTY

Example: PUNCH@EMPTY ON SORTFILE ONLY SORT

Explanation: The EMPTY modifier may be applied to the EDIT, EXPLAIN, LIST, and PUNCH commands to request that the output file is to be emptied before output from the macro-library editor is written to it.

Modifier: FULL

Example: LIST@FULL MINER

Explanation: The FULL modifier may be appended to the ADD, COPY, INCLUDE, LIST, REPLACE, and UPDATE commands. The FULL modifier specifies that the entire macro is to be listed.

Modifier: HDRGEN

Example: INCLUDE@¬HDR@NAME=GDINFO *GDINFODSECT

Explanation: The HDRGEN modifier specifies that the macro-library editor should supply the copy-section name header in form '*COPY SECTION name', where "name" is specified by the NAME modifier.

Modifier: INCREMENT=n

Example: ADD@INCREMENT=20 HISFILE

Explanation: This INCREMENT modifier is used in the ADD, COPY, INCLUDE, RENUMBER, REPLACE, or UPDATE commands to set the increment line number. The default is 1.000.

Modifier: LIST

Example: UPDATE@¬LIST MAC

Explanation: The LIST modifier may be appended to the ADD, COPY, INCLUDE, REPLACE, and UPDATE commands. Macro names, their beginning and ending line numbers, and their line counts are listed. The VERIFY option is overridden by the LIST modifier.

Modifier: NAME=symbol

Example: ADD@NAME=GDINFO *GDINFODSECT

Explanation: The NAME modifier is used in the ADD, COPY, INCLUDE, REPLACE, or UPDATE commands to specify that the input file is a copy section with name "symbol".

Modifier: SEQ

Example: ADD@NOSEQ OSMAC ONLY DXR

Explanation: SEQ specifies that macro-library editor is to retain the sequence ID in columns 73 through 80. NOSEQ requests that the editor should remove the sequence IDs.

Modifier: SORT={LNR|NAME}

Example: LIST@SORT=LNR@FULL

Explanation: The SORT modifier may be appended to the LIST command. Macros to be listed are first sorted according to their macro names in alphabetic order (SORT=NAME) or according to their beginning line numbers in ascending order (SORT=LNR). The default is SORT=NAME.

Modifier: START=lnr

Example: RENUMBER@START=500

Explanation: The START modifier may be specified in the RENUMBER command so that the first source line starts at the line number "lnr". The default is the START parameter (usually 1000.).

Modifier: TERSE

Example: REPLACE@TERSE FROM IMPROVEDMACS

Explanation: The TERSE modifier may be applied to some commands to abbreviate the information produced for verification. VERBOSE is an antonym of TERSE. The default is the setting of the MTS TERSE option.

Modifier: VERBOSE

Example: UPDATE@VERBOSE FROM NEWMACROS

Explanation: If TERSE has been turned on globally via SET TERSE=ON, then full information for the verification of a particular command can be produced via the VERBOSE modifier. VERBOSE is an antonym of TERSE. The default is the setting of the MTS VERBOSE option. This modifier has no effect if verification is suppressed via the @¬VERIFY modifier or the command SET VERIFY=OFF.

Modifier: VERIFY

Example: ADD@V -LOAD

Explanation: If verification has been turned off globally via SET VERIFY=OFF, then it can be enabled for a particular command via the VERIFY modifier.

PRINCIPLES OF OPERATION

MACUTIL determines if the file being edited is a macro library in the following way. The file specified should be a line file with no explicit beginning, ending, or increment MTS line numbers. If the file is found to be empty, MACUTIL will consider it to be an empty macro library. Otherwise, it will attempt to read a line at 1.000. If there is no such line, the file is not a macro library. But if there is, that line, plus successive lines, until "00000000" at columns 1-8 is encountered, are called directory lines. If the file is a valid macro library, all directory lines should conform to the following rules:

(1) First column should be alphabetic.
(2) There should be two fields each terminated by a blank.
(3) The first field starts at column 1 and must only consist of not more than 80 alphanumeric characters. This field indicates the macro name.
(4) The second field is separated from the first field by a series of blanks. This designates a valid MTS line number for the beginning line of the macro.

If any of the directory lines do not conform to the rules, MACUTIL indicates that the file being edited is not a macro library. If all directory lines are valid, the line number associated with EOD (end of directory, with eight zeros at columns 1-8) is remembered. The next line read is considered as the first source line; all directory lines will always precede this line. MACUTIL then sorts the directory lines in ascending order according to the beginning line numbers. Each beginning line number is checked. If it is not greater than the EOD line number, an error message is produced and the directory line is discarded. If the beginning line number is not integral, a warning will be produced but MACUTIL will nevertheless accept the directory line. Since the directory lines are already sorted with their beginning line numbers in ascending order, MACUTIL proceeds to determine the ending line number of a macro in the following way. The beginning line number of the next macro, less 0.001, is first calculated (if the macro is the last macro, the actual last line number of the file will be set, instead). MACUTIL then reads backwards until it encounters "$ENDFILE" or MEND. It will also stop reading backwards whenever it discovers the line number is less than the beginning line number. Once the ending line number is found, all lines between the beginning and ending line numbers are counted. If the count is zero, an error message is produced and the directory line discarded.

The next section describes the operation of MACUTIL when including macros or copy sections from an external file in the ADD, COPY, INCLUDE, REPLACE, and UPDATE commands. It also uses the same method of scanning when it is building a directory for an existing macro library except that only macro names with their associated beginning line numbers are placed in the directory.

If the NAME modifier was specified on the command, the (copy section) name in the directory is set to that name. A copy-section header is generated according to the HDRGEN modifier. The NAME modifier is then blanked out to prevent any more copy sections with the same name. MACUTIL then sequentially reads lines of the file until it either encounters "$ENDFILE" or the actual end of the file. "$ENDFILE" is automatically generated, if the "$ENDFILE" trailer is not present. If "$ENDFILE" was encountered, MACUTIL will then proceed to process any remaining macros or copy sections in the file being included in the manner described below (i.e., as if the NAME modifier was not specified).

If the NAME modifier was not specified on the command, MACUTIL will scan the file being included for "MACRO" ... "MEND" pairs or "*COPY SECTION name" ... "$ENDFILE" pairs. If COMSAVE is ON and "MACRO" was found, MACUTIL will also include any preceding comments (lines that start with an "*" or ".*"). For each pair, it generates a directory entry and automatically inserts "$ENDFILE" after "MEND". A line is a copy-section header if columns 1-5 contain "*COPY", column 6 a blank, a hyphen, or an underscore, columns 7-14 "SECTION ", and remaining columns specify the copy section name, terminated by a blank.

MACUTIL utilizes an alternate method whenever the program discovers the external file being included is a macro library. This is indicated by the presence of a valid directory line number 1.000. In this case, the NAME modifier is invalid and ignored. MACUTIL (selectively if an slist is specified) includes macros and copy sections by interrogating the directory of the external file. MACUTIL must check if it is including a macro or a copy section. It will be a macro if the prototype statement's operation code matches the macro name; otherwise, MACUTIL is including a copy section. MACUTIL reads all lines of a macro until MEND, matching the first MACRO is encountered. For all copy sections, "$ENDFILE" terminates a copy section.

As an illustration, consider the file being included by the ADD command is:

```
                ...
                MACRO
                XXX
                ...
                MEND
                ...
                MACRO
                YYY
                ...
                MEND
                ...
*COPY SECTION AAA
                ...
$ENDFILE
                ...
```

```
        *COPY SECTION BBB
                    ...
        $ENDFILE
                    ...
```

The macro library will be:

```
           1.       XXX      1000
           2.       YYY      1100
           3.       AAA      1200
           4.       BBB      1300
           5.       00000000
        1000.               MACRO
        1001.               XXX
                            ...
        1009.               MEND
        1010.       $ENDFILE
        1100.               MACRO
        1101.               YYY
                            ...
        1109.               MEND
        1110.       $ENDFILE
        1200.       *COPY SECTION AAA
                            ...
        1210.       $ENDFILE
        1300.       *COPY SECTION BBB
                            ...
        1310.       $ENDFILE
```

May 1983

MACRO-LIBRARY EDITOR EXAMPLE

A sample terminal run is given below to illustrate several of the features of the macro-library editor. In this example, user input is in lowercase while macro-library editor output is in uppercase. Note also that the macro-library editor uses two slashes (//) as the prompting prefix.

```
#$run *macutil
#EXECUTION BEGINS
 MACUTIL VERSION(PR148) 09:40:22 04-17-78
//*
//*comment - Example #1: to set up a macro library from macros
//*
//edit -mac
//display status
 Macro library "-MAC" has 0 macros and 0 lines.
//add minimac
 ADDED:
  MIN      MINR     MINH     MINL     MINLR    MINE     MINER
  MIND     MINDR
//display directory

 --- 9 macros ---

 MIN      MIND     MINDR    MINE     MINER    MINH     MINL
 MINLR    MINR
//*
//*comment - Example #2: to extract macros from a macro library
//*
//add@full from *sysmac only mts

 Macro MTS: 6 lines added

    1900              MACRO
    1901     &SLB     MTS
    1902     &SLB     L     15,=V(MTS)
    1903              BALR  14,15
    1904              MEND
    1905     $ENDFILE
//delete mine
 DELETED:
  MINE
//update@terse from minimac
 REPLACED:
 8 macros
 ADDED:
 1 macro
//display directory

  --- 10 macros ---
```

```
  MIN      MIND     MINDR    MINE     MINER    MINH     MINL
  MINLR    MINR     MTS
//*
//comment - Example #3: another way to extract a macro
//*
//edit *sysmac
 *** WARNING: The macro library "*SYSMAC" cannot be used for output.
//display status
 Macro library "*SYSMAC" has 170 macros, 1 copy section,
 and 5684 lines.
//list spie
 Macro SPIE (5600.,5745.) 146 lines
//punch on -spie only spie
 PUNCHED:
  SPIE
//*
//comment - Example #4: to build a directory in the file being edited
//*
//buildir -mac
 *** ERROR: The macro library file "-MAC" already has the directory.
 A new directory is to be rebuilt. OK?   ok
 Done.
//display status
 Macro library "-MAC" has 10 macros and 59 lines.
//list@full mine

 Macro MINE (2000.,2005.) 6 lines

    2000              MACRO
    2001     &LABEL   MINE  &RA,&SB
    2002     &LABEL   CE    &RA,&SB
    2003              BNH   *+8
    2004              LE    &RA,&SB
    2005              MEND
//list
 Macro MIN (1000.,1005.) 6 lines
 Macro MIND (1700.,1705.) 6 lines
 Macro MINDR (1800.,1805.) 6 lines
 Macro MINE (2000.,2005.) 6 lines
 Macro MINER (1600.,1605.) 6 lines
 Macro MINH (1200.,1205.) 6 lines
 Macro MINL (1300.,1305.) 6 lines
 Macro MINLR (1400.,1405.) 6 lines
 MACUTIL ATTN
//stop


 CPU time = 1.81 seconds.
#EXECUTION TERMINATED      09:50.05 T=1.912   $.62
```

May 1983

ASSIST ASSEMBLER AND INTERPRETER

This section is intended to serve as a basic reference guide for programmers using Assembler Language and the ASSIST assembler/interpreter system for the IBM 370/168, Amdahl 470V/6, or equivalent computer running under MTS. ASSIST (Assembler System for Student Instruction and Systems Teaching) is a small, high-speed, low-overhead assembler/interpreter system especially designed for use by students learning assembler language. The assembler program accepts a large subset of the standard Assembler language, and includes most commonly used features. The execution-time interpreter simulates the full 370 instruction set, with complete checking for errors, meaningful diagnostics, and completion dumps of much smaller size than the normal system dumps.

This section describes the necessary commands for running ASSIST, the assembly language commands and instructions permitted by the ASSIST assembler, error messages given by both the assembler and the interpreter, and gives other useful information.

It is assumed that the reader is already familiar with the IBM Assembler/370 language and the basic MTS commands. A basic familiarity with the following two publications is also assumed: IBM System/360 Principles of Operation (form GA22-6821) and IBM System/370 Principles of Operation (form GA22-7000).

ASSIST follows the basic IBM linkage conventions. That is, when the user's ASSIST program is entered, GR13 points to an 18-word save area, GR1 points to a count and parameter list, and the program is called by a BALR 14,15.

Parts of the following manuals have been used in the preparation of this section:

ASSIST Introductory Assembler User's Manual, John R. Mashey, Pennsylvania State University, August 1971.

ASSIST User's Guide, W. C. Jackson, University of Alberta, Edmonton, Alberta, Canada, January 1973.

UBC STASS, The Stass Assembler, P. Campbell, I. Raudzins, W. Dettwiler, and L. Horvath, University of British Columbia Computing Centre, Vancouver, British Columbia, Canada, November 1971.

RUNNING ASSIST UNDER MTS

THE MTS $RUN COMMAND AND LOGICAL UNIT SPECIFICATIONS

The command to run ASSIST is:

$RUN *ASSIST [logical I/O unit assignments] [PAR=parameters]

where the logical unit assignments are as follows:

SCARDS — the file or device for assembler input (i.e., control cards, source deck, and data). The default is *SOURCE*.

SPRINT — the file or device for assembler output (i.e., source listing, diagnostic messages, program results, and job accounting information). The default is *SINK*.

SERCOM — the file or device for a few terminal error messages. The default is *SINK*.

0,2,3 — macro library files, formatted as per *MACGEN. See the section, "ASSIST Macro Libraries," later in this volume for further details. In order to use a macro library, the user must provide a *SYSLIB card.

CONTROL CARDS

All the control cards described below must have their text beginning in column one of the input card to be recognized as a control card. They may all be abbreviated by their first 4 alphabetic characters.

/ASSEMBLE

This control card may be used to identify the beginning of an assembler program for ASSIST to assemble (and possibly execute). Its presence is required only between separate assembler programs being assembled under a single $RUN of ASSIST (see the BATCH parameter and examples later in this volume). However, it may appear at the beginning of any assembler program to identify the program and set local assembly options before its assembly and execution.

The control card contains a comment field and a parameter field. The comment field may consist of any string of characters, excluding a colon (:), and follows the "/ASSEMBLE" after one or more intervening blanks. The comment field is terminated by a

colon, after which an optional list of compiler parameters may appear (see the "Parameters" section below for their description). For example,

```
/ASSEMBLE  FIRST PROGRAM CHECKOUT : TIME=.2,LIBMC
```

/EXECUTE

This control card may optionally be present at the end of the assembler program source deck. A parameter string may be specified following the "/EXECUTE". This will then be passed to the student's program via the standard conventions (i.e., GR1 points to a halfword location, which contains the length of the string, followed by the PAR string exactly as given on the /EXECUTE card). For example,

```
/EXECUTE   PAR=YES,CHECK
```

would cause GR1 to contain the address of a halfword location containing 9 (the length of the string), followed by the characters YES,CHECK. The first blank after the PAR= terminates the string.

/DATA

This control card may be used in place of /EXECUTE.

/STOP

This control card may be used at the end of a batch to terminate ASSIST. An end-of-file on SCARDS behaves in the same manner, so /STOP is optional in this case.

PARAMETERS

The ASSIST parameters listed in the table on the following pages are of two types. The first group consists of on/off type parameters and are negated by placing 'NO' in front of them. The second group of parameters requires assigned values.

Parameters may be set from two different locations. The first is via the PAR= field on the $RUN command; the second is after a colon (:) on the /ASSEMBLE card. If specified on the $RUN command, they apply to all jobs in a batch, whereas the parameters specified on the /ASSEMBLE card apply only to that particular job. Certain parameters may be specified only on the $RUN card, and not on the /ASSEMBLE card. These are flagged by an asterisk (*) in the table. Others can be reset, but never to a value higher than a previously set value. These are flagged by a plus sign (+) in the table. (See Key at end of table.)

The parameters may be specified in any order. Some examples follow:

```
$RUN *ASSIST SCARDS=*SOURCE* PAR=NOLIST,PX=5,BATCH
```

This command would suppress the source listing, set a limit of 5 on the number of pages of output allowed during execution, and permit batching of jobs.

```
/ASSEMBLE JOHN DOE :NOLIST,PX=5
```

This card would again suppress the source listing and limit the number of pages of output allowed during execution to 5. However, if, on the $RUN command, 'PAR=PX=2' had been specified, the 'PX=5' on the /ASSEMBLE card would have no effect since PX cannot be set to a value higher than the value previously set.

Table of ASSIST Parameters

Parameter	Default	Maximum	Description
*BATCH	NOBATCH		Allows more than one job to be run at a time.
CMPRS	NOCMPRS		Gives a compressed source listing of 2 columns/page.
COMNT	NOCOMNT		Requires 80% of the source statements to have a comment.
*CPAGE	CPAGE		Forces control of paging.
LIBMC	NOLIBMC		Macros fetched from the library will be printed.
LIST	LIST		Produces assembly source listing.
LOAD	LOAD		Produces object code and runs it.
SS	NOSS		Singlespaces assembly (if CPAGE is turned on).
SSD	NOSSD		Singlespaces dump (if CPAGE is turned on).
SSX	NOSSX		Singlespaces execution (if CPAGE is turned on).
*SIZE=	15		Indicates the amount of virtual memory to be used for working storage. This is effectively the only limit on the size of the program which can be assembled.
KP=	029		Type of keypunch used (026 or 029).
+L=	60	66	Number of lines per page if CPAGE is turned on.
MACTR=	200		Default value for starting ACTR counters in all macros used. Can be overridden by explicit ACTR statements.
MNEST	15		Limits level of nested macro calls, thereby preventing unwanted recursion in macros.

Parameter	Default	Maximum	Description
MSTMG=	4000		Limits total number of statements processed in all macro expansions. Another caution against looping macros.
NERR=	0		Maximum number of assembly errors permitted, before execution is disallowed.
+P=	25	100	Number of pages for the job-assembly, execution, and dump (if CPAGE is turned on).
+PAGES=	10	100	Number of pages allowed for execution (if CPAGE is on).
+PD=	2	100	Number of pages to be saved for a dump (if CPAGE is turned on).
+PX=	10	100	Number of pages allowed for execution (if CPAGE is turned on).
+T=	10	100	Time allowed for the entire job-assembly, execution, and dump.
+TD=	0.5	10	Time to be saved for the dump.
+TIME=	10	100	Time allowed for execution.
+TX=	10	100	Time allowed for execution.
XREF=	0		Gives a cross-reference of the variables. To get a simple cross-reference, set XREF= 2. A detailed description on how to use this parameter will be found in a later section.

Key

* -parameters that may be specified only on the $RUN card.
+ -parameters which may be reset, but never to a higher level than previously set.

May 1983

SAMPLE DECK SETUPS

The following example is for a single run, and is the most common way of running ASSIST.

```
$RUN *ASSIST [PAR= parameters]
/ASSEMBLE [comments] [:parameters] (may be omitted)
   source cards
/EXECUTE [PAR= parameter string] (may be omitted)
   data (if any)
/STOP      (may be omitted)
$ENDFILE
```

The following would be the setup for a batch of assemblies:

```
$RUN *ASSIST PAR=BATCH [,parameters]
/ASSEMBLE [comments] [:parameters] (may be omitted)
   source cards for program 1
/EXECUTE [PAR= parameter string]
   data for program 1  (if any)
/ASSEMBLE [comments] [:parameters] (required)
   source cards for program 2
/EXECUTE [PAR= parameter string]
   data for program 2 (if any)
   .
   .
   .
/STOP      (may be omitted)
$ENDFILE
```

THE ASSEMBLY LANGUAGE UNDER ASSIST

This part deals with the subset of the standard MTS/370 Assembler Language accepted by the ASSIST assembler. Headings and subheadings in this part have been taken directly from the IBM publication GC28-6514, IBM System/360 and 370 Operating System Assembler Language. Because the standard is followed very closely, only those language features which ASSIST omits or treats differently are described here. Users should consult the IBM Assembler/370 reference manuals for most of the information on the Assembler Language.

INTRODUCTION

Macro Instructions

Many system macros are available (see later section). Macros may also be programmer defined. Many execution time services are provided by the interpreter through system subroutines (see later section); macros are provided to call these subroutines.

The Assembler Program

The assembler program produces a listing of the source program, and creates an object program directly in main memory, using no secondary storage devices. An object deck cannot be punched.

GENERAL INFORMATION

General Restrictions on Symbols

A symbol may be defined only once in an assembly (i.e., it may appear in the name field of no more than one statement). The same symbol may not be used as a label in two different control sections and control sections may not be resumed. In the standard language, this is the only case where the same symbol is allowed on more than one statement.

Location Counter Reference

ASSIST allows full use of the location counter *, with the following exceptions:

(1) The programmer may not refer to the location counter inside a literal address constant. Thus, the following statement will produce incorrect results:

```
L 1,=A(*+20)
```

(2) The programmer may not refer to the location counter in an A-type address constant having a duplication factor greater than one, if the reference is made in such a way that the various duplications of the specified constant have different values. For instance, under Assembler/370, the following statement would produce the values 0,1,...,255, but ASSIST would produce 256 bytes of zero:

```
NAME     DC    256AL1(*-NAME)
```

Literals

Literal constants may not contain more than 112 characters, counting the beginning equal sign (=) and the ending delimiter (i.e., literal constants may not require more than two cards when placed in the literal pool).

Literal Pool

Unless otherwise specified by the use of the LTORG instruction, the literal pool is placed after the program's END card, rather than at the end of the first control section in the program.

Expressions

Use of general expressions is permitted for most statements. Any restrictions are noted under the individual statements.

ADDRESSING - PROGRAM SECTIONING AND LINKING

USING - Use Base Register

The first expression (address) in a USING statement must be relocatable. However, the use of a base register in ASSIST is not always necessary.

Control Sections

Multiple control sections are allowed. A program must contain at least one control section.

Control Section Location Assignment

Control sections may not be intermixed under ASSIST (i.e., all the statements of one control section must be coded before another is begun).

First Control Section

Under ASSIST, the first control section has no properties different from the other sections (i.e., its initial location counter value must be relocatable), and it does not normally contain unassigned literal constants unless it is the only control section.

START - Start Assembly

The START instruction may be preceded by listing control instructions and comment cards. The same label may not be used on a START statement and a later CSECT statement. START may be used to name the first (or only) control section of a program.

CSECT - Identify Control Section

No more than one CSECT may use a given symbol as a name, and statements from different CSECTs may not be interspersed.

DSECT - Identify Dummy Section

No more than one DSECT may use a given symbol as a name, and statements from different DSECTs may not be interspersed.

External Dummy Sections (Assembler F only)

External dummy sections are not supported, so the commands CXD and DXD are not recognized.

COM - Define Blank Common Control Section

COM is not allowed.

MACHINE-INSTRUCTIONS

Instruction Alignment and Checking

If any statement requires alignment and causes bytes to be skipped, the bytes skipped are NOT set to hexadecimal zeros.

Operand Fields and Subfields

ASSIST permits the same use of expressions in machine-instruction operand fields as does the standard assembler, with one restriction: registers may be specified by single absolute symbols, by single self-defining terms (B, X, C), or by decimal numbers within the range 0 to 15, having no leading zeros. Thus, general expressions may not be used to specify registers.

ASSEMBLER LANGUAGE STATEMENTS

OPSYN - Equate Operation Code

This statement is not accepted.

DC - Define Constant

Multiple operands (up to 10 operands in a single DC statement), and multiple constants within operands are both permitted. Bytes skipped to align a DC statement are NOT zeroed.

Operand Subfield 3: Modifiers

The following modifiers are not permitted by ASSIST: Bit-Length Specification, Scale Modifier, and Exponent Modifier.

Operand Subfield 4: Constant

Fixed-Point Constants: - F and H

Fixed-point constants may not contain decimal points or exponents. While lengths may range from one to eight bytes, the minimum and maximum values permitted are those for length 4.

Floating-Point Constants: - E and D

No scale or exponent modifiers are allowed, but exponents are accepted within each constant.

Decimal Constants: - P and Z

If no explicit length is supplied for an operand containing multiple constants, each of the operands is assembled to the length of the last constant in the operand, even if truncation is thus required. Under the standard assembler, for example, the following needs four bytes:

```
     DC    P' 0,20,1'
```

Under ASSIST, it is assembled into three bytes, with the second constant truncated.

Address Constants

Only A and V address constants are allowed.

Complex Relocatable Expressions

These are not allowed.

A-type Address Constant

Adcons may not be used in a literal constant if it refers to the location counter. It will be assembled improperly if it does so.

Y-Type, S-Type, and Q-Type Address Constants

These are not allowed.

CCW - Define Channel Command Word

The CCW is recognized and allocated storage, but is not otherwise assembled. It will be flagged 'NOT CURRENTLY IMPLEMENTED'.

Listing Control Instructions

TITLE - Identify Assembly Output

No title may have a symbol in the name field.

PRINT - Print Optional Data

All operands are accepted, but DATA and NODATA have no effect (i.e., no more than eight bytes of data are ever printed). Any statement flagged with an error or warning is always printed, even if the print control is OFF, or NOGEN for generated statements is in effect.

Program Control Instructions

The following four control instructions are not accepted by ASSIST:

ICTL - input format control.
ISEQ - input sequence checking.
PUNCH - punch a card.
REPRO - reproduce following card.

LTORG - Begin Literal Pool

Any literals used after the last LTORG are placed after the END card, instead of at the end of the first control section. Duplicate literals are never stored, since programmers may not refer to the location counter in a literal A-type address constant; this is the only case under the regular system that requires the storing of duplicate literals.

COPY - Copy Predefined Source Coding

COPY is not allowed.

INTRODUCTION TO THE MACRO LANGUAGE

ASSIST allows macros compatible with Assembler (F). These macros may be either programmer-defined macros, or macros included from the system macro library, or both. The use of macros from the ASSIST system macro library requires a special comment card, *SYSLIB. See the section on "Macro Libraries" later.

Open Code Conditional Assembly

With certain restrictions (noted below), ASSIST will allow the use of conditional assembly statements and SET variables outside macros (i.e., in the open code, or main body of the program).

The Macro Definition

COPY statements are not allowed.

System and Programmer Macro Definitions

System macros are described in a later section. Macros may also be programmer defined.

HOW TO PREPARE MACRO DEFINITIONS

Macro Instruction Prototype

Two formats are allowed for statements, the normal one used by all other statements, and the alternate one allowed only for macro prototype and macro call statements. ASSIST does allow macro prototypes and macro calls to be continued on an indefinite number of cards. When there are no more than two continuation cards, ASSIST is completely compatible with other assemblers. If the total number of cards in a statement exceeds three, the following restriction must be observed: every third card in the statement must use the alternate format, unless it is the last one. (This is done because ASSIST processes cards in groups of three.) The two prototypes which follow illustrate this restriction:

PROTOTYPE ACCEPTED BY ASSEMBLERS F, G, H, VS, BUT NOT ASSIST:

```
&LABEL  LONGPROT  &PARM1,&PARM2,      PARMS,ALTERNATE FORMAT      X
                  &PARM3,&PARM4,&PARM5,  PARMS,ALTERNATE FORMAT      X
                  &PARM6,&PARM7=XXXXXXX,&PARM8=YYYYYYYYYY,&PARM9=ZZ,&X
                  PARM7=A                LAST LINE
```

EQUIVALENT PROTOTYPE, ACCEPTED BY ASSIST:

```
&LABEL  LONGPROT  &PARM1,&PARM2,     PARMS,ALTERNATE FORMAT       X
                  &PARM3,&PARM4,&PARM5, PARMS,ALTERNATE FORMAT       X
                  &PARM6,&PARM7=XXXXXXX,&PARM8=YYYYYYYYYY,&PARM9=ZZ, X
                  &PARM7=A              LAST LINE
```

Given this restriction, it is best to place early in the list any positional parameters that may require long values needing continuation.

Model Statements

Variable symbols may be used to generate the operation field inside macros only. One extension is that they are allowed to generate PRINT and END instructions if desired.

Copy Statements

COPY statements are not allowed.

May 1983

HOW TO WRITE MACRO INSTRUCTIONS

There are no changes from the IBM standard.

HOW TO WRITE CONDITIONAL ASSEMBLY INSTRUCTIONS

All of the conditional assembly instructions may be used inside macros. They may be used outside, but there are restrictions as given below.

Attributes

ASSIST is a two-pass assembler, performing macro-processing during pass 1. Consequently, it is usually impossible for it to know the attribute of a symbol, so there are definite restrictions. In effect, the only attributes that are allowed in a conditional assembly instruction are those which can be found by looking at a macro call statement itself. The attributes allowed are listed as follows:

Attribute	Notation	
Type	T'	only values N, O, and U possible
Count	K'	
Number	N'	

Thus, Length (L'), Scaling (S'), and Integer (I') attributes are not supported in conditional assembly instructions. The only values for Type are N (Numeric), O (Omitted), and U (Undefined), so that in many cases the value is U under ASSIST where it would be something else under IBM assemblers.

AIF - Conditional Branch

IBM assemblers normally assign 4096 as the usual limit for the number of AIF and AGO branches. See ACTR for a description of the manner in which ASSIST handles this.

The sequence symbol named in the AIF may precede or follow the AIF statement inside macros. Outside macros, it may only follow AIF (i.e., only forward branches are allowed). If a branch is taken to a previously defined sequence symbol in open code, ASSIST does not know that it was defined, and so will read to the END card, skipping the rest of the program.

AGO - Unconditional Branch

AGO follows the same restriction as AIF: backwards branches are allowed in macros, but not in open code.

ACTR - Conditional Assembly Loop Counter

ASSIST supports the standard ACTR. However, the default value of the ACTR counter is set differently, via the MACTR= option supplied by the user. This has a default value which is normally smaller than the IBM default value of 4096. The MACTR= value is used for all macro definitions, unless explicitly overridden via ACTR statements.

Conditional Assembly Elements

There are no changes, except that attributes L', S', and I' are not supported.

EXTENDED FEATURES OF THE MACRO LANGUAGE

MNOTE - Request for Error Message

The MNOTE statements accepted by ASSIST follow the standard, but ASSIST effectively ignores the use of severity codes. MNOTEs with a numerical severity code are printed as errors, while those with an asterisk (*) are printed in another format.

&SYSECT

CSECT or DSECT statements processed in a macro definition do not affect the value of &SYSECT for any subsequent inner macros in their definition.

Macro Definition Compatibility

ASSIST does not accept AGOB or AIFB.

EXECUTION-TIME SERVICES

ASSIST provides execution-time services in three ways. First, additional mnemonics are provided which allow input/output statements. Second, a type of supervisor call is provided which gives various dumps, more input/output services, and a method of termination. Third, a group of system subroutines which allow for various types of services is provided by the interpreter.

INPUT/OUTPUT MNEMONICS

Eight input/output instructions have been added to the instruction set to provide the facility for reading a single card or writing a single line of integer, real, or alphanumeric data according to fixed formats. In terms of implementation, the instructions use a modified version of the FORTRAN format scan and conversion routines. The eight instructions are all of the SI type and the mnemonics are as follows:

Input

RI A,n

reads n (n≤8) fullword integers into the location starting at A.

FORMAT: (8I10)

RHI A,n

as above, but reads halfword integers.

REI A,n

reads n (n≤8) fullword real numbers into the location starting at A.

FORMAT: (8G10.0)

RCI A,n

reads n (n≤80) consecutive alphanumeric characters into the n consecutive bytes starting at A.

FORMAT: (80A1)

Output

WI A,n

writes n (n≤8) fullword integers from the location starting at A.

FORMAT: (1X,8I15)

WHI A,n

as above, but writes halfword integers.

WEI A,n

writes n (n≤8) fullword real numbers from the location starting at A.

FORMAT: (1X,8G15.6)

WCI A,n

writes n (n≤120) consecutive alphanumeric characters from the location starting at A.

FORMAT: (1X,120A1)

Notes

Following execution of an input instruction, the condition code is set to:

0 - when an end-of-file has been encountered (no read performed, so A remains unchanged), or to
3 - when the read operation was successful.

The condition code is unchanged by execution of an output instruction.

As in FORTRAN (MTS), input data fields may be shortened with a comma on the data card. For example, the data card for the instruction

RI A,2

could be

1,2,

with the '1' in column 1 of the card. The commas then have the effect of reducing the input field from 10 columns (I10) to 1 column (I1).

May 1983

SUPPLEMENTARY MNEMONICS

SUP 0

Normally this will terminate the simulation of the current program, and return control to the monitor.

SUP 1

This will terminate the simulation of the current program, cause a register and memory dump, and return control to the monitor.

SUP 2

This will print the contents of the general and floating-point registers in hexadecimal, and then continue with the following instruction.

SUP 3

This will print the contents of the general and floating-point registers in decimal, and then continue with the following instruction.

SUP 4

This will print the words located between two 4-byte addresses in hexadecimal as follows. The setup is:

```
SUP   4
DC    A(A)
DC    A(B)
```

where A is the starting address and B is the ending address of the area to be printed. Execution then continues with the first statement past the DCs.

SUP 5

This will read a line under a FORTRAN-type FORMAT. The setup is:

```
SUP   5
DC    A(FORMAT)
DC    AL1(COUNT)
DC    AL3(ARRAY1)
DC    AL1(COUNT)
DC    AL3(ARRAY2)
.
.
.
DC    X'FF'
```

where FORMAT is the address of a format. For example, in the following:

```
      FORMAT    DC    C'(2I5,5F5.2)'
```

'COUNT' is the number of items in the ARRAY specified by the address in the following instruction. To simplify the specification of single-word items, a count of zero is equivalent to a count of 1. 'FF' in the count position terminates the data list.

Condition code settings are:

0 - if an end-of-file has been read
3 - if a normal read occurred.

Note that all addresses specified for the I/O list must be fullword items. Hence halfword integers, doubleword real numbers, or alpha-numeric strings which are not fullword-aligned may give disastrous results. Note also that L- and D-type format specifiers are not allowed in the format.

SUP 6

This will print a line under a FORTRAN-type FORMAT. The setup is the same as in SUP 5, but the condition code remains unchanged. Note however, that a carriage-control character must be provided, but if it is nonblank, it is set to zero.

SYSTEM SUBROUTINES

The following pages contain a description of system services available through the following subroutines:

CTI
ERROR
FREESPAC
GETSPACE
ITC
READ - SCARDS
SYSTEM
TOD
TROFF
TRON
WRITE - SPRINT

The subroutines listed above require standard MTS linkages. The subroutine names are not reserved names, i.e., they may be used as labels or variable names; however, if this is done, they may not be used as V-type external references. For example, if the user has a statement labeled WRITE, then V(WRITE) will be flagged as an invalid external reference. If the symbol WRITE was an external symbol defined by the

program (by CSECT or ENTRY), then V(WRITE) would be resolved to the program entry point WRITE, not to the system subroutine WRITE.

Note that when using these subroutines, unusual address locations may show up in dumps, traces, error messages, etc. These have the form of "FFxx" where "xx" will be two hexadecimal numbers. These are used for the internal addresses of the subroutines listed above.

CTI

Function

To convert a character string to a binary fullword integer.

Parameters

GR0 - length of character string in bytes.
GR1 - address of the character string.

Return Value

GR2 - result of character-to-integer conversion.

Error Exits

A data exception will occur if there are invalid characters in the string (not +, -, 0-9), or if the string is invalid (such as having more than one sign). A data exception will also occur if the number is not in the range $-2^{31} \leq X \leq 2^{31}-1$. A protection exception will occur if the string, as defined by GR1 and GR0, is not within the user's program. A specification exception will occur if the length in GR0 is greater than 10.

Description

The specified character string is checked for errors as explained above. If the string is valid, conversion is performed and the result is placed in GR2. If the conversion fails, a dump is given and the contents of GR0 and GR1 are unchanged.

Example

```
        .
        .
        .
        L    GR0,LENGTH   LOAD STRING LENGTH
        LA   GR1,STRING   LOAD ADDR OF STRING
        L    GR15,=V(CTI) LOAD EP ADDRESS
        BALR GR14,GR15    LINK TO CTI
        ST   GR2,RESULT   SAVE INTEGER ANSWER
        .
        .
        .
```

May 1983

ERROR

Function

To cause the program to terminate abnormally and produce a dump of the state of the program.

Parameters

None.

Return Value

None - the subroutine never returns.

Error Exits

None.

Description

The program terminates with actions identical to those of SUP 1; i.e., a register dump, a memory dump, and a branching trace history will be displayed.

Example

```
        .
        .
        .
LTR   15,15               NON-ZERO RETURN CODE?
BZ    OK                  IT'S ZERO - EVERYTHING'S FINE
L     15,=V(ERROR)        NONZERO - MUST HAVE AN ERROR SOMEPLACE
BALR  14,15
```

FREESPAC

Function

To release an area of main storage obtained with a GETSPACE subroutine call.

Parameters

GR1 contains the address of the area to be released. (This address must be the same as the area address supplied in GR1 after a successful GETSPACE subroutine call.)

Return Value

GR15 - return code
0 - free complete
4 - free unsuccessful

Error Exit

None.

Description

This subroutine allows the user to release an area of main storage dynamically obtained with the GETSPACE subroutine. When FREESPAC is called, GR1 must contain the address of an allocated area. If the address in GR1 is not the same as the address returned on a previous successful call to GETSPACE, a return code of 4 is given, indicating that the specified region has not been released.

Example

```
        .
        .
        .
        L    GR1,COREADDR       LOAD AREA ADDR
        L    GR15,=V(FREESPAC)  LOAD EP ADDR
        BALR GR14,GR15          LINK TO SUBR
        LTR  GR15,GR15          TEST RET CODE
        BNZ  NOTFREED           BR IF NOT FREED
        .
        .
        .
```

GETSPACE

Function

To obtain space dynamically in main storage.

Parameters

GR1 contains the amount of storage required in bytes.

Return Value

GR15 - return code
   0 - request successful
   4 - request unsuccessful

If the request was successful:

GR1 - contains the address of the first byte of the allocated area. The first fullword of the allocated area is set to the area's length.

If the request was unsuccessful, GR1 is unchanged.

Error Exit

None.

Description

A request for main storage is made by placing the number of bytes required in GR1 and calling GETSPACE. Upon return, GR15 will contain a return code. If the return code is 4, it indicates that the requested amount of storage was not available, and the contents of GR1 is unchanged. If the return code is 0, the amount of storage requested has been allocated. Storage is only allocated in double-word increments starting on doubleword boundaries. GR1 will contain the area address, and the first fullword of the area will contain the amount of storage allocated (always a multiple of 8 bytes).

Example

```
        .
        LA   GR1,200
        L    GR15,=V(GETSPACE) LOAD EP ADDR
        BALR GR14,GR15         LINK TO SUBR
        LTR  GR15,GR15         TEST RET CODE
        BNZ  NOGOTIT           BR IF NOSPACE
        MVC  AMOUNT(4),0(GR1) SAVE AMOUNT
        ST   GR1,COREADDR      SAVE LOCATION
        .
```

ITC

Function

To convert a binary integer to a character string.

Parameters

GR0 - contains the length of character string in bytes.
GR1 - contains the address of first byte of string where the result will be placed.
GR2 - contains the integer to be converted to a character string.

Error Exits

A specification error will occur if the length in GR0 is greater than 256 bytes. A protection exception will occur if the string as defined by GR1 and GR0 is not in the user's program.

Description

The binary integer contained in GR2 is converted to a character string and is placed in the area defined by GR1 and GR0. If necessary, the string is right-justified and padded with blanks on the left-hand side. If the converted string is too large for the defined area, the area is filled with asterisks.

Example

```
        .
        .
        .
L    GR0,LENGTH   LOAD STRING LENGTH
LA   GR1,STRING   LOAD ADDRESS OF STRING
L    GR2,NUMBER   LOAD INTEGER
L    GR15,=V(ITC) LOAD EP ADDRESS
BALR GR14,GR15    LINK TO ITC
        .
        .
        .
```

READ, SCARDS

Function

To read a card into the specified area.

Parameters

GR1 - load with address of the input buffer.

Return Value

GR0 - contains the length of the data read. This is always 80.
GR15 - return code
0 - successful read
4 - end-of-file

Error Exit

If the address specified is outside the program, a protection or addressing exception is indicated, and the program is terminated with a dump.

Description

A card is read and placed in the specified area. If there are no remaining data cards, an end-of-file condition is indicated and the contents of the input buffer remain unaltered. A subsequent attempt to read past the end-of-file will cause the user's program to be terminated abnormally. Note that the buffer will always be padded with blanks to fill the 80-byte area.

Example

```
      .
      .
      .
CARD  DS   CL80           INPUT BUFFER
      .
      .
      LA   GR1,CARD       LOAD BUFFER ADDR.
      L    GR15,=V(READ)  GET EPA
      BALR GR14,GR15      LINK TO READ
      LTR  GR15,GR15      TEST RETURN CODE
      BNZ  ENDFYLE        BRANCH ON EOF
      .
      .
      .
```

or

```
      .
      .
      .
CARD  DS   CL80           INPUT BUFFER
      .
      .
      LA   GR1,CARD       LOAD BUFFER ADDR.
      L    GR15,=V(SCARDS) GET EPA
      BALR GR14,GR15      LINK TO SCARDS
      LTR  GR15,GR15      TEST RETURN CODE
      BNZ  ENDFYLE        BRANCH ON EOF
      .
      .
      .
```

SYSTEM

Function

To terminate execution of the program in a normal fashion.

Parameters

None.

Return Values

None; the subroutine never returns.

Error Exits

None.

Description

The action of this subroutine is identical to that of the execution of the SUP 0 instruction; the program terminates execution.

Example

```
        .
        .
        .
L     15,=V(SYSTEM)       TH-TH-THAT'S ALL, FOLKS!
BALR  14,15
```

TOD

Function

To obtain time of day and date as a character string.

Parameters

None.

Return Value

GR0 and GR1 - contain the time of day
GR2 and GR3 - contain the date

Error Exit

None.

Description

The time is in the form

HH:MM.SS

where HH, MM, and SS are hours, minutes, and seconds, respectively. For example, 23:14.33 indicates the time as being 14 minutes and 33 seconds past 11 p.m.

The date is in the form

MM-DD-YY

where MM, DD, and YY are month, day, and year, respectively. For example, February 23, 1970 is indicated 02-23-70.

Example

```
          .
          .
          .
TIMEDATE DC   4F'0'  SAVE TIME/DATE HERE
          .
          .
          L    GR15,=V(TOD)     LOAD EP ADDR
          BALR GR14,GR15        LINK TO TOD
          STM  GR0,GR3,TIMEDATE SAVE RESULTS
          .
          .
          .
```

May 1983

TROFF

Function

To turn off branching trace.

Parameters

None.

Return Value

None.

Error Exit

None.

Description

This subroutine is used to stop the branch trace initiated by subroutine TRON. If trace is off, this subroutine has no effect.

Example

```
       .
       .
       .
       L    GR15,=V(TROFF) LOAD EP ADDR.
       BALR GR14,GR15      LINK TO TROFF
       .
       .
       .
```

TRON

Function

To turn on branching trace.

Parameters

None.

Return Value

None. Return is to the address in general register 14.

Error Exit

None.

Description

Every time a branch instruction is executed, a line of format is printed as follows:

BRANCH from XXXX to XXXX MM YYYYYYYY

Where: XXXX are hexadecimal addresses
MM is the assembler mnemonic, e.g., BC, BCT, ...
YYYYYYYY is the branch instruction in hexadecimal.

Notes

1. This subroutine does not change the condition code.

2. This should be used with care, as every branch executed after returning from this subroutine will cause a line to be printed until TROFF is called. Therefore, it is very easy to exceed the page limit.

3. If trace is on and this subroutine is called, it has no effect.

Example

```
      .
      .
      .
      L    GR15,=V(TRON) GET EP ADDR.
      BALR GR14,GR15     LINK TO TRON
      .
      .
      .
```

WRITE, SPRINT

Function

To write a line of specified length from the specified buffer.

Parameters

GR1 - contains the address of output buffer.
GR0 - contains the length of output buffer in bytes.

Return Value

None.

Error Exit

If the specified buffer is outside the program, a protection or addressing exception is indicated and the program is terminated with a dump. If the specified length in GR0 is not in the range 1 to 133, the program is terminated with an error message and a dump.

Description

A line of specified length is written taking the data from the specified area. The first byte of the buffer is used as carriage control.

Carriage Control

C'1' - skip to top of a logical page before printing.
C' ' - space 1 line before printing.
C'0' - space 2 lines before printing.
C'-' - space 3 lines before printing.
C'+' - print without spacing.
C'2' - skip to next 1/2 page before printing.
C'4' - skip to next 1/4 page before printing.
C'6' - skip to next 1/6 page before printing.
C'8' - skip to bottom of logical page before printing.
C'9' - print single-space and suppress logical page overflow.
C';' - skip to top of physical page before printing.
C'>' - skip to bottom of physical page before printing.

Note: If the "CPAGE" option is in effect, only the carriage controls "1", " ", "-", and "+" are recognized; all others cause single-spacing to be performed.

Example

```
        .
        .
        .
  LINE DC   C'1PAGE SKIP' OUTPUT BUFFER
        .
        .
        .
        LA   GR0,L'LINE     LOAD BUFFER LENGTH
        LA   GR1,LINE       LOAD ADDR OF LINE
        L    GR15,=V(WRITE) LOAD EP ADDR.
        BALR GR14,GR15      LINK TO WRITE
        .
        .
        .
        LA   GR0,L'LINE     LOAD BUFFER LENGTH
        LA   GR1,LINE       LOAD ADDR OF LINE
        L    GR15,=V(SPRINT) LOAD EP ADDR.
        BALR GR14,GR15      LINK TO SPRINT
```

OUTPUT AND ERROR MESSAGES

ASSEMBLY LISTING

Assembly Listing Format

The assembly listing produced by the ASSIST assembler is essentially the same as that produced by the standard MTS/370 Assembler, with the following minor differences:

(1) Error messages are not printed at the end of the assembly listing, but are printed after the statement causing the message. A scan pointer '$' indicates the column where the error was discovered.

(2) No more than four messages are printed for any single source statement. Some errors cause termination of statement scan, and errors following in the same statement may not be discovered. However, an error in a statement does not normally prevent its statement label from being defined, which is usually the case with the standard assembler.

(3) Statements that are flagged are printed regardless of print status at the time.

(4) As noted under PRINT earlier, no more than eight bytes of data are printed for a statement, even if PRINT DATA is used.

Assembler Error Messages

The assembler produces error messages consisting of an error code followed by an error description. The code is of the form AS###, with the value of ### indicating one of three types of errors.

(1) Warnings - ### is in the range 000-099. These never prevent the execution of the program, and have messages beginning with characters 'W-'.

(2) Errors - ### is in the range 100-899. Execution is deleted if the total number of errors exceeds the NERR parameter.

(3) Disastrous errors - ### is in the range 900-999. Some condition prevents successful completion of the assembly process. Execution of the user program may or may not be permitted.

The following is a list of the codes and messages issued by the ASSIST assembler.

AS000 W-ALIGNMENT ERROR-IMPROPER BOUNDARY
The address used in a machine instruction is not aligned to the correct boundary required by the type of instruction used.

AS001 W-ENTRY ERROR-CONFLICT OR UNDEFINED
A symbol named in an ENTRY statement is either undefined, or is named in a DSECT or EXTRN statement.

AS002 W-EXTERNAL NAME ERROR OR CONFLICT
A symbol named in an EXTRN statement is either defined in the program or is named in an ENTRY statement.

AS003 W-REGISTER NOT USED
The register flagged in a DROP statement is not available for use as a base register at this point in the program. This may be caused by an error in a USING statement naming the register.

AS004 W-ODD REGISTER USED-EVEN REQUIRED
An odd register is coded in a machine instruction requiring the use of an even register for a specific operand. Instructions which may be flagged are Multiply, Divide, Double Shifts, and all floating-point instructions.

AS005 W-END CARD MISSING-SUPPLIED
The assembler creates an END card because the user has supplied none before an end-of-file occurred.

AS100 ADDRESSABILITY ERROR
An implied address is used which cannot be resolved into base displacement form. No base register is available which satisfies the following conditions: first, it must contain an address which is less than, but in the same CSECT as, the implied address. Second, the implied address must fall within 4095 bytes of the address in the base register.

AS101 CONSTANT TOO LONG
Too many characters are coded for the type of constant specified. This message appears if a literal constant contains more than 112 characters, including the equal sign and delimiters.

AS102 ILLEGAL CONSTANT TYPE
An unrecognizable type of constant is specified.

AS103 CONTINUATION CARD COLS. 1-15 NONBLANK
A continuation card contains nonblank characters in columns 1-15. This may be caused by an accidental punch in column 72 of the preceding card.

AS104 MORE THAN 2 CONTINUATION CARDS
Three or more continuation cards are used, which is illegal.

AS105 COMPLEX RELOCATABILITY ILLEGAL
ASSIST does not permit complex relocatable expressions.

AS106 TOO MANY OPERANDS IN DC
ASSIST allows no more than ten operands in a DC statement.

AS107 MAY NOT RESUME SECTION CODING
The assembler requires that any section be coded in one piece. The label flagged has already appeared on a CSECT or DSECT.

AS108 ILLEGAL DUPLICATION FACTOR
A duplication factor exceeds the maximum value of 32,767; in a literal constant, the duplication factor has a value of zero or is not specified by a decimal term.

AS109 EXPRESSION TOO LARGE
The value of the flagged expression or term is too large for the given usage. For example, a constant length is greater than the maximum permissible for the type of constant.

AS110 EXPRESSION TOO SMALL
The value of the flagged expression or term is too small for the given usage, or has a negative value. Coding a V-type constant with a length of two would generate this message.

AS111 INVALID CNOP OPERAND(S)
The operands of a CNOP have values which are illegal combinations of values for a CNOP, such as a first operand greater than the second, an odd value, etc. The only legal value combinations are 0,4 2,4 0,8 2,8 4,8 6,8.

AS112 LABEL NOT ALLOWED
A label is used on a statement which does not permit one, such as a CNOP or USING statement.

AS113 ORG VALUE IN WRONG SECTION OR TOO LOW
The expression in an ORG statement has either a value smaller than the initial location counter value for the current control section, or has a relocatability attribute different from that of the current control section.

AS114 INVALID CONSTANT
A constant contains invalid characters for its type, or is specified improperly in some other way.

AS115 INVALID DELIMITER
The character flagged cannot appear where it does in the statement. This message is used whenever the scanner expects a certain kind of delimiter to be used, and it is not there.

AS116 INVALID FIELD
The field flagged has an unrecognizable value, or is otherwise incorrectly coded. PRINT OFF is flagged this way.

AS117 INVALID SYMBOL
The symbol flagged either contains nine or more characters or does not begin with an alphabetic character as required.

AS118 INVALID OP-CODE
The statement contains an unrecognizable mnemonic op-code, or no op-code.

AS119 PREVIOUSLY DEFINED LABEL
The symbol in the label field has been previously used as a label.

AS120 ABSOLUTE EXPRESSION REQUIRED
A relocatable expression is used where an absolute one is required, such as in constant duplication factor or for a register.

AS121 MISSING DELIMITER
A delimiter is expected but not found. For instance, a C-type constant coded with no ending prime (') is flagged this way.

AS122 FEATURE NOT CURRENTLY IMPLEMENTED
The version of ASSIST being used does not support the language feature used.

AS123 MISSING OPERAND
The instruction requires an operand, but it is not specified.

AS124 LABEL REQUIRED
An instruction requiring a label, such as a DSECT, is coded without one.

AS126 RELOCATABLE EXPRESSION REQUIRED
An absolute expression or term is used where a relocatable one is required by ASSIST, such as in the first operand of a USING statement. This message may also appear if the final relocatability attribute of the value in an address constant is that of a symbol in a DSECT.

AS127 INVALID SELF-DEFINING TERM
The self-defining term flagged contains an illegal character for its type, has a value too large for 24 bits to contain, or is otherwise incorrectly specified.

AS128 ILLEGAL START CARD
One or more statements, other than listing controls or comments, appear before the START card.

AS129 ILLEGAL USE OF LITERAL
The literal constant appears in the receiving field of an instruction which modifies memory. For example, ST 0,=F'1'.

AS130 UNDEFINED SYMBOL
The symbol shown is either completely undefined, or has not been defined when required. Symbols used in ORG instructions, in constant lengths, or in duplication factors must be defined before they are used.

AS131 UNRESOLVED EXTERNAL REFERENCE
The symbol used in a V-type constant is not defined in the assembly, or, it is defined but not declared a CSECT or ENTRY. ASSIST does not link multiple assemblies, so this is an error.

AS132 ILLEGAL CHARACTER
The character flagged is either used in an illegal way, or is not in the set of acceptable characters.

AS133 TOO MANY PARENTHESES LEVELS
Parentheses are nested more than five deep in an expression.

AS134 RELOCATABLE EXPRESSION USED WITH * OR /
Relocatable terms or expressions may not be used with either of these operators.

AS135 SYNTAX
The character flagged is improperly used. This catchall message is given by the general expression evaluator when it does not find what is expected during a scan.

AS136 TOO MANY TERMS IN EXPRESSION
The expression contains more than the legal maximum of 16 terms.

AS137 UNEXPECTED END OF EXPRESSION
The expression terminates without having enough right parentheses to balance the left parentheses that were used.

The following messages are issued only during macro processing:

AS201 OPERAND NOT ALLOWED
During macro expansion, an extra operand was found, i.e., an extra positional parameter beyond those given in the prototype.

AS202 STATEMENT OUT OF ORDER
The statement flagged is in an incorrect place in the deck. For example, LCLx before GBLx, ACTR after both, or GBLx, LCLx, ACTR in middle of macro definition or open code. This is often caused by a missing MEND card.

AS203 SET SYMBOL DIMENSION ERROR
A dimension set symbol was used without a dimension, or one which was not dimensioned was written with a dimension.

AS204 INVALID NBR OF SUBSCRIPTS
There was an error in specifying substring notation, sublists, or set symbol dimension.

AS205 ILLEGAL CONVERSION
During macro editing, a SET instruction was found with an obviously incorrect conversion, as in the following:

```
&I    SETA  C
```

AS206 MISSING QUOTES IN CHAR EXPR
Single quotes were not supplied as required in character expressions.

AS207 ILLEGAL OR DUP MACRO NAME
A macro prototype name is either completely illegal (i.e., has too many characters), or duplicates the name of a previously given macro, machine instruction, or assembler instruction.

AS208 OPRND NOT COMPATIBLE WITH OPRTR
An operand is used with an incompatible operator. For example, if &C is LCLA, &B LCLB ; &B SETB (NOT &C).

AS209 UNDFND OR DUPLICATE KEYWORD
In calling a macro, a keyword is used which does not appear in the macro prototype. In defining or calling a macro, a keyword operand appears twice or more in the list of operands.

AS210 MNEST LIMIT EXCEEDED
The MNEST option provides a maximum limit to the nested depth of macro calls. This limit has been exceeded. Note that after the MSTMG limit has been exceeded, the MNEST is effectively 0.

AS211 ILLEGAL ATTRIBUTE USE
ASSIST does not support S', I', or L' for macro operands.

AS212 GENERATED STATEMENT TOO LONG
A statement having more than two continuation cards was generated.

AS217 STMT #### NOT PROCESSED BECAUSE OF PREV ERROR
During expansion, the statement numbered #### was ignored, because it was already flagged with a previous error.

AS218 STORAGE EXCEEDED BY FOLLOWING MACRO EXPANSION
The following call to the macro listed caused overflow of storage, probably due to looping. Use ACTR, MACTR=, or MSTMG= .

AS220 UNDEFINED SEQUENCE SYMBOL IN STATEMENT #####
This may appear following an entire macro definition, and gives the number of a statement referencing a sequence symbol which was never defined.

The following messages may be issued during macro expansion. The #### gives the number of a statement in some macro definition in which an error has occurred during expansion. Some messages also display an appropriate value, such as an offending subscript. Note that the messages below use ## as an abbreviation for the actual output, which is normally printed by ASSIST in the form STMT/MACRO ####/name.

AS221 STMT #### ACTR COUNTER EXCEEDED ##
The ACTR count has been exceeded. The ACTR is set by the MACTR option, or by an ACTR statement. This indicates looping within a macro.

AS222 STMT #### INVALID SYM PAR OR SET SYMBOL SUBSCRIPT ## --> value
A subscript is out of range. The offending value is given.

AS223 STMT #### SUBSTRING EXPRESSION OUT OF RANGE ## --> value
This is most often caused by the first subscript in a substring expression having a nonpositive value, or by a subscript larger than the size of the string.

AS224 STMT #### INVALID CONVERSION, CHAR TO ARITH ## --> value
The value could not be converted to an arithmetic form.

AS225 STMT #### INVALID CONVERSION,ARITH TO BOOLEAN ## --> value
The value was not 0 or 1.

AS226 STMT #### INVALID CONVERSION, CHAR TO BOOLEAN ## --> value
The value was not '0' or '1', so it could not be converted.

AS227 STMT #### ILLEGAL ATTRIBUTE USE ##
An attribute was used incorrectly.

AS228 STMT #### &SYSLIST SUBSCRIPT OUT OF RANGE ##
The subscript has a value greater than the maximum number of fields which can be supplied.

AS229 STMT #### CALL FRIENDLY ASSIST REPAIRMAN ##
Internal error; please bring your input deck and the output to the attention of a member of the Computing Center staff.

AS230 STMT #### INTERNAL CHAR BUFFER EXCEEDED ##
Too much concatenation was done in the statement. Reduce the complexity of the statement.

AS231 STMT #### MSTMG LIMIT EXCEEDED ##
The MSTMG limit (total number of statements processed during macro expansion) has been exceeded. Use MSTMG= to increase this.

AS232 STMT #### ZERO DIVIDE OR FIXED POINT OVERFLOW ##
One of these interrupts was caused by the statement given.

AS241 FOLLOWING SEQUENCE SYMBOL NOT FOUND --> symbol
During the processing of an AIF or AGO in open code, an END card was encountered before the sequence symbol. Either the label was not defined or a backwards branch was being attempted (not allowed).

AS242 BACKWARDS AIF/AGO ILLEGAL
This message immediately follows an AGO or successful AIF in open code which references a previously defined sequence symbol. ASSIST allows backwards branches only in macros, not in open code.

AS288 MACRO xxxxxxxx COULD NOT BE FOUND
This is issued by the macro library processor when it tries to get a macro and cannot find it in the library. The macro may be named on a *SYSLIB card, or referenced by another macro.

AS289 UNABLE TO OPEN MACRO LIBRARY: OPTION CANCELLED
This is issued after a *SYSLIB card is encountered, but the macro library cannot be opened. Please bring your input deck and the output to the attention of a member of the Computing Center staff.

AS298 GENERATED STMTS OVERWRITTEN
During macro expansion, one or more generated statements were lost due to internal table management, probably because a statement near the beginning of a macro generated a long literal constant. One solution is to insert several comment cards at the beginning of the macro definition.

AS999 DYNAMIC STORAGE EXCEEDED
The dynamic storage area available to ASSIST has been exceeded, so that assembly cannot proceed. Reassemble with a larger SIZE parameter value.

Assist Monitor Error Messages

The ASSIST monitor may also issue one of the following messages, which are of the form AM###, and usually indicate errors:

AM003 STORAGE OVERFLOW BEFORE EXECUTION, EXECUTION DELETED
The user program assembled properly, but there is insufficient memory remaining to set up control blocks required for execution. The user should attempt to reduce the amount of storage used by his program. This message should seldom occur.

AM005 TIME OR PAGES HAVE BEEN EXCEEDED
This message is printed if the time or page limits have been exceeded at any time, other than execution, during a job.

Assembler Statistics Summary

Following the assembly listing, the assembler prints three or four lines of statistical information, as follows:

*** ##### STATEMENTS FLAGGED - ##### WARNINGS, ##### ERRORS

This notes the total number of statements flagged, warning messages, and error messages given during the assembly.

***** NUMBER OF ERRORS EXCEEDS LIMIT OF ##### ERRORS - PROGRAM EXECUTION DELETED *****

This tells the user that the maximum number of errors, NERR, which may occur before execution will be inhibited, has been exceeded.

*** DYNAMIC CORE AREA USED: LOW: ###### HIGH: ###### LEAVING: ###### FREE BYTES. AVERAGE: ###### BYTES/STMT ***

The ASSIST assembler acquires a single memory area at execution time. The LOW area is used to store source statements and generated object code, the HIGH area is used to store the symbol and literal tables. The space remaining indicates how close the program is to causing a storage overflow. The average use of memory printed includes that used in both LOW and HIGH areas.

*** ASSEMBLY TIME = #.### SECS. ##### STATEMENTS/SEC ***

This notes the total time used by the assembler, along with the rate of assembly.

EXECUTION PHASE - THE INTERPRETER

The interpreter executes the given object code by interpreting each instruction separately, thus trapping any program interrupt that may occur. If execution is possible after an interrupt, a message as to the cause and the action taken is given, and the interpretation continues. Four interrupts may occur for each type of interruption, after which the interrupt is treated as an abnormal end and execution is terminated. The last ten branch instructions are then printed, and the registers and memory are dumped.

The user can trace branching in a specified area of his or her program, via the TRON and TROFF subroutines (described previously), and can request dumps of registers or areas of memory via special supplementary instructions.

Type of Printout

Explanation of Symbols Used

hhhh	indicates the current relative address in hexadecimal.
pppp	is the program mask in binary.
c	indicates the condition code, 0, 1, 2, or 3.
xxxxxxxx	indicates the hexadecimal instruction or contents of a register or word.
dddddddd	indicates the contents of a register in decimal.
s	indicates the sign, either a blank for plus or a minus.
aaaa	indicates the hexadecimal address from which a core dump is to be printed.
bbbb	indicates the hexadecimal address up to which a core dump is to be printed.
xxxx	indicates the hexadecimal address on a doubleword boundary.
hhha	indicates the hexadecimal address.
hhhb	indicates the hexadecimal address.
l	is the instruction length in halfwords.
ic	is the interrupt code in decimal.
000000n	indicates the number of branches executed so far, in decimal with leading zeros.
mmmm	is mnemonic for a branch instruction.

May 1983

Supplementary Calls

SUP 0

This terminates the execution with the message:

END OF EXECUTION NUMBER OF INSTRUCTIONS EXECUTED = n.

SUP 1

This produces an abnormal termination, which causes the program to stop executing, and a dump of the following items to be printed:

(1) the current location, program mask, and condition code;
(2) the last 10 branches executed by the program;
(3) a register dump in hex;
(4) a register dump in decimal;
(5) a dump of all core associated with the assembled program, including any GETSPACE allocations.

SUP 2

```
REGISTER DUMP (HEX) AT LOCATION hhhh PROGRAM MASK= pppp CONDITION  CODE=
c

GPR 0  xxxxxxx   xxxxxxx   . . . . . .
GPR 8 xxxxxxx      .      . . . .
FPR 0    xxxxxxxxxxxxxxxx     FPR 2  x.  .  .  .  .
```

SUP 3

```
REGISTER  DUMP (DEC) AT LOCATION hhhh PROGRAM MASK= pppp CONDITION CODE=
c

GPR 0 sddddddddd   .  .  .  .  .  .
GPR 8 sddddddddd   .  .  .  .  .  .
FPR    s.ddddddddddddddddDsdd    .  .  .  .  .
```

SUP 4

```
CORE DUMP  AT  LOCATION  hhhh  PROGRAM  MASK=  pppp  CONDITION  CODE=  c
LOCATIONS aaaa TO bbbb

xxxx      xxxxxxxx     xxxxxxxx   .  .    .  .  .  .
(xxxx+8)      etc.
```

If a multiple of 8 words is the same, the following line is printed:

```
WORDS hhha TO hhhb CONTAIN xxxxxxxx
```

Execution-Time Error Messages

All the messages are one line long and one of two types:

**** WARNING AT LOCATION hhhh. Reason for error.
INSTRUCTION action. PROGRAM CONTINUED.

'action' is either:

TERMINATED - The result may or may not be correct, depending at which stage the interrupt causing the error was found.

NOT EXECUTED - Nothing is altered.

COMPLETED - The instruction is executed, probably giving an erroneous result.

**** FATAL ERROR AT LOCATION hhhh. Reason for error. PROGRAM TERMINATED.

This is followed by:

*ABNORMAL END AT LOCATION hhhh. PROGRAM MASK= pppp CONDITION CODE= c INSTRUCTION LENGTH= l INTERRUPT CODE= ic.

The last message is also invoked by SUP 1, or when the allowed number of warnings has been exceeded. This is followed by:

LAST 10 BRANCHES (IN REVERSE ORDER OF EXECUTION)
TOTAL NUMBER OF BRANCHES = 000000n

and then a minimum of n or 10 branch instructions of the form

BRANCH FROM hhha TO hhhb mmmm xxxxxxxx

The register dumps in hexadecimal and decimal then follow as described under SUP 2 and SUP 3, respectively. Finally, the memory dump is given in the same format as described under SUP 4. The entire object program is printed in hexadecimal. Note that memory is not zeroed before the object code is generated and areas such as those defined by a DS may contain junk. The program is terminated as described under SUP 0.

Each error message has an associated interrupt code. The only time the code is printed is when an abnormal termination occurs. At other times the message is printed.

00 ABNORMAL END
Invoked by SUP 1.

01 INVALID OPERATION
The current instruction address points to a location which contains an invalid operation code.

02 PRIVILEGED OPERATION
An operation which is privileged (e.g., LOAD PSW) has been encountered in the problem program state. All SVCs are privileged in ASSIST.

03 EXECUTE SUBJECT OF EXECUTE

04 CORE REFERENCED OUTSIDE OF USER'S AREA
This corresponds to the protection exception. The memory specified by the operand of an instruction lies outside the area reserved for the user's program. See the IBM Principles of Operation manuals.

05 ADDRESSING EXCEPTION
The address specified is outside of the available machine storage.

06 INCORRECT BOUNDARY ALIGNMENT, REG., I/O COUNT
This corresponds to the specification exception. An operand is not on a proper boundary for the instruction being executed, or the length for the CTI or ITC subroutine is incorrect.

06 DEC ARITH - MULT OR DIV LENGTH SPECS
Either the multiplier or divisor in decimal arithmetic exceeds 15 digits and sign or the first operand field is shorter than or equal to the second operand field in decimal multiplication or division.

06 ERROR IN ARGUMENT TO SYSTEM SUBROUTINE
This arises when the arguments presented to one of the system subroutines are not of the proper size, type, or alignment. Check the subroutine description and the code calling the subroutine for bad parameters.

07 DEC ARITH - DIGIT CODES OR FIELD DEFINITION
This corresponds to the data exception. The sign of digit codes of operands in decimal arithmetic or editing operations or in CONVERT TO BINARY are incorrect, fields in decimal arithmetic overlap incorrectly, or the decimal multiplicand has too many high-order significant digits.

08 FIXED POINT OVERFLOW
A high-order carry occurs, or high-order significant bits are lost in fixed point add, subtract, shift, or sign-control operation.

09 FIXED POINT DIVIDE
A quotient exceeds the register size in fixed point division, including division by zero.

09 CONVERT TO BINARY EXCEEDS 31 BITS

10 DECIMAL OVERFLOW
The destination field is too small to contain the result field in a decimal operation. (The operation is completed by ignoring the overflow information.)

11 DECIMAL DIVIDE
A quotient exceeds the specified data field size. (The operation is suppressed.)

12 EXPONENT OVERFLOW
The result characteristic in floating-point addition, subtraction, multiplication, or division exceeds 127 and the result fraction is not zero. (The operation is completed. The fraction is normalized, and the sign and fraction of the result remain correct. The result characteristic is made 128 smaller than the correct characteristic.)

13 EXPONENT UNDERFLOW
The result characteristic in floating-point addition, subtraction, multiplication, halfing, or division is less than zero, and the result fraction is not zero. (The operation is completed. The setting of the exponent-underflow mask (PSW BIT 38) effects the result of the operation. When the mask bit is zero, the sign, characteristic, and fraction are set to zero, making the result a true zero. When the mask bit is one, the fraction is normalized, the characteristic is made 128 larger than the correct characteristic, and the sign and fraction remain correct.)

14 SIGNIFICANCE - ALL-ZERO EXPONENT FRACTION
The result of a floating-point addition or subtraction has an all-zero fraction. (The operation is completed.)

15 FLOATING-POINT DIVIDE
Division by a floating-point number with zero fraction is attempted. (The operation is suppressed.)

16 INVALID REG-MULT, DIV,DBL,SHFT,FLT PT ARITH
The R field of an instruction specifies an odd register address for a pair of registers that contain a 64-bit operand (e.g., divide, shift double instruction, etc.) Or a floating-point register other than 0, 2, 4, or 6 is specified.

17 BRANCH OUT OF USER'S AREA
This corresponds to the protection exception and causes a fatal error.

18 BRANCH TO ODD LOCATION
This corresponds to the specification exception and causes a fatal error. The branch instruction specifies an address that is not on a halfword boundary.

19 TIME OR PAGES EXCEEDED
The time or pages allowed has been exceeded during execution and an abnormal termination occurs.

22 YOU RAN OUT OF DATA
An attempt has been made to read past the end of data indication. Result is an abnormal termination.

23 I/O LIST ADDRESS OUTSIDE PROGRAM
The address for the I/O list in the SUP 5 or SUP 6 instruction is outside the user's program area.

24 INVALID TYPE OF FORMAT
The FORMAT specified in the SUP 5 or SUP 6 instruction is not valid. Abnormal termination results.

25 LINE TOO LONG, BAD FORMAT
May be caused by SUP 5 or 6 and also by the WRITE or SPRINT subroutines, if the length given is not between 1 and 133 inclusive.

26 INVALID CHARACTER IN DATA
The data to be input or output contain an invalid character.

27 INVALID HEXADECIMAL DATA
Data being read or written with a hexadecimal FORMAT contain invalid hexadecimal characters.

28 INVALID I/O LIST ADDRESS ALIGNMENT
SUP 5 or SUP 6 address list pointers are not multiples of 4.

THE CROSS-REFERENCE OPTION

This option provides a short, but informative cross-reference listing following the assembly listing. Besides noting where every symbol is defined in the assembly, it distinguishes between two types of references. A modify reference is one in which a symbol is used in a machine instruction field denoting an operand to be modified: ST 0,X for example. All other references are considered fetch references: B X, L 0,X, DC A(X). The cross-reference output shows a symbol, its value, and statement numbers of referencing statements, with modify references flagged as negative statement numbers. Control of the output is obtained both by the XREF= option, and by *XREF cards inserted in the source program as desired. The latter permit explicit control of how references are gathered.

A brief note on the XREF mechanism is necessary to make use of the flexible control provided. During Pass 1 of an assembly, the SD (Symbol Definition) flag is attached to each symbol as it is defined. The flag consists of two bits (M for Modify and F for Fetch, in that order), and shows what kinds of references may possibly be collected for each symbol. For example, SD=10 indicates that no fetch references are ever to be printed for a specific symbol. The SD flag may be changed during a program by *XREF cards, so that symbols in different sections of the program can be treated differently: SD=00 will eliminate all following symbols completely, until it is changed again.

During Pass 2, a Symbol Reference (SR) flag is used to determine what types of references are being collected from the code. A reference to a symbol is logged if and only if the SD bit and the SR bit for the given type of reference are both on. For example, if SD=10 for a symbol, SR=11 at the current time, and a fetch reference is made, no reference will be logged, since the SD fetch bit is 0. Note that references are only logged during Pass 2: some symbol references occur only during Pass 1, and these are ignored, such as symbols in EQU, ORG, and DC and DS length modifiers or duplication factors.

The XREF parameter requests a cross-reference, indicates the type of output produced, and possibly gives initial values to the SD and SR flags. Two forms are permitted as follows:

XREF=a or XREF=(a,b,c)

a - indicates overall control and output format.
  = 0: no cross-reference is generated.
  = 2: cross-reference is printed, with one symbol per output line.
  = 3: cross-reference is printed, but with minimal output wasted (more than one symbol may appear on a line -- this form is recommended).

b - indicates initial value of SD flag in decimal corresponding to

binary (i.e., 0: 00, 1: 01, 2: 10, 3: 11).
c - indicates initial value of SR flag, same format as b.

Illegal values are ignored, and it is allowable to omit items as desired, showing this by comma usage: XREF=(2,,2) for example. The default value is XREF=(0,3,3) so that all that is needed to obtain a complete listing is to code XREF=2 or XREF=3, as the other values are not changed or zeroed.

The SR and SD flags may be changed at any time during the program, by placing *XREF comment cards anywhere in the source program following the first machine instruction or assembler opcode used (SD options used before these will work, but SRs will be ignored). The format is:

*XREF [SD=nn] [SR=nn]

The operands(s) may be specified in any order, and if the same option is used several times, requested actions are performed in order. The options are:

SD=<M><F> give the modify and fetch bits for the SD flag.
SR=<M><F> give the modify and fetch bits for the SR flag.

Possible values for <M> and <F> are:

0 - turn bit off.
1 - turn bit on.
* - leave bit in previous state.

If an <F> specification is omitted, this is equivalent to a *.

It is suggested that the user begin by simply specifying XREF=2 or 3 and then cutting out unnecessary references later. Although complex, the facilities allow unwanted output to be eliminated easily. The following gives an example (assumed to be a large program):

*XREF	SD=10	following symbols will have only modify references.
.....	large number of DS and DC statements (global table, for example).	
*XREF	SD=*1	add modify and fetch references both.
.....	more symbols in DSECTS, tables, etc.	
*XREF	SD=00,SR=10	collect no references to symbols defined from here on, collect and modify references created.
.....	section of code which is referencing tables above.	
*XREF	SD=11,SR=11	collect all references from following code to second part of table, modify references to

first part and all references to itself.

..... section of code which is referencing tables.

ASSIST MACRO LIBRARIES

SOURCES OF MACRO LIBRARIES

ASSIST assemblies may draw on macros defined in any of three places: the system macro library (*ASSISTMAC), the beginning of the assembler program, or a private macro library. Macros whose definitions appear at the beginning of the assembly do not need to be brought to the attention of ASSIST in any special manner.

However, ASSIST must be notified of macros whose definitions are to be obtained from system or private libraries. Private macro libraries should be in line files adhering to the standard assembly library format (as produced by *MACUTIL), and should be attached to the logical I/O units 0, 2, or 3 by their specification on the $RUN *ASSIST MTS command (see the earlier section, "Running ASSIST Under MTS"). Whenever the macro libraries are searched for a macro definition, the libraries attached to logical I/O units 2, 3, and 0 are searched, in that order, for the macro; its first instance terminates the search. If unit 0 is not assigned, *ASSISTMAC is implicitly assigned to it. Therefore, any program using only the system macro library need not specify the assignment.

THE *SYSLIB CARD

It is desirable that users specify whether the macro library should be searched; this prevents searching automatically for a misspelled opcode name in the library. A special comment card, *SYSLIB, is used to inform ASSIST that it should actually perform a library search, and lists those system macros which are referenced in the open code. The format of the *SYSLIB card is:

```
*SYSLIB       name1,name2,...  comments
```

where "name1,name2,..." is a list of the names of each system macro, separated by commas in free format.

The *SYSLIB card should follow all programmer macros (if any), and must precede any of the statements of the open code, except for comment and listing control (PRINT, TITLE, EJECT, SPACE) statements. The user may supply one or more *SYSLIB cards, as long as these conditions are fulfilled.

When finding any *SYSLIB card in a proper location, ASSIST does the following:

(1) Scans the card, adding any name found there to the list of macro names. If the name is already in the list, it is totally ignored.

(2) Scans the list of macro names. If a macro is not defined, it searches the macro library for it. If the macro cannot be obtained, it internally marks the macro as "searched for," and never looks for it again.

(3) If the macro is found during step 2, the print control is turned OFF, unless the user specified LIBMC, in which case the print control is unchanged. The macro is then read and edited, like a programmer macro.

(4) During step 3, the macro being read may refer to other macros not yet defined, and these are added to the macro list also. The loop of steps 2,3,4 continues until all macros in the list have either been found or searched for. Thus, it is possible for a reference to one macro to cause a number of macros to be fetched from the library. At this point, print control is restored to its original value, and a list of undefined macros is produced.

The following gives the overall layout of a program:

..... 0 or more programmer macro definitions, with print control statements interspersed if desired.
..... 1 or more *SYSLIB cards.
..... 0 or more GBLx declarations.
..... 0 or more LCLx declarations.
..... ACTR.
..... Open code (main body of program).

The following shows appropriate *SYSLIB use, although the program itself should not be expected to make sense:

```
            MACRO
            MYCALL &PARA
            CALL   &PARA
            MEND
  *SYSLIB   SAVE,RETURN        CALL WILL AUTOMATICALLY BE INCLUDED
            USING  *,15
            SAVE   (14,12)
            MYCALL TRON
            RETURN (14,12)
            END
```

May 1983

HINTS ON OPTIMAL USE OF A MACRO LIBRARY

The user should be aware of the following when using the macro library facility:

(1) The macro processor is mainly intended to process programmer-written macros. Among other things, all macro dictionaries and tables are kept in memory because it is faster than keeping them in files.

(2) Many of the macros have inner macro expansions which cause more than one macro to be brought in from the library.

(3) If a macro is referenced, it is fetched from the library, whether or not it is actually ever called.

(4) ASSIST operates in a fixed size work space that must contain all macros, tables, and source code. To allow the assembly of large programs in this small area, the user must avoid the excessive use of macros which consume a large amount of space.

MACRO DESCRIPTIONS

The following macros are available in the system macro library supported by ASSIST, *ASSISTMAC. These macros allow compatibility with some MTS macros found in *SYSMAC, and will generate the code required by ASSIST to perform the equivalent functions. This will allow a program to be run under ASSIST and then be assembled under *ASMG by making use of the macros in *SYSMAC.

ENTER

Purpose

To generate a subroutine prolog which:

(1) Saves general registers.
(2) Establishes a base register.
(3) Establishes a new save area and provides forward and backward pointers linking save areas.

Prototype

[label] ENTER reg[,TREG=tempreg] [,SA=savearea] [,LENGTH=length]

where:

reg is used as a base register by the subroutine.

tempreg is a register used by the generated code. If omitted, R15 is used.

savearea is the location of an 18-word save area. If omitted, a save area is dynamically obtained via GETSPACE.

length is the length in bytes of the save area obtained via GETSPACE. If omitted, the length is assumed to be 72; otherwise, it should be at least 72.

Comments

The generated prolog assumes that the subroutine is entered, with register 15 containing the address of the entry point. The call to GETSPACE uses registers 1, 14, and 15, so that they cannot be used as a base register if the save area is obtained dynamically. If the length parameter is coded, it must be at least 72 or an addressing exception will result. A SUP 1 is generated if the GETSPACE is unsuccessful.

reg and tempreg must specify different registers. Neither reg nor tempreg may specify registers 0 or 13.

Examples

```
ENTER 12,SA=MYSAVE
ENTER 11,TREG=8,LENGTH=200
ENTER 10,LENGTH=100
ENTER 9
```

EXIT

Purpose

To generate a subroutine epilog which:

(1) Restores general registers.
(2) Reestablishes the calling program's save area.
(3) Sets the return code in register 15.
(4) Sets value to be returned in register 0.

Prototype

```
[label] EXIT  [rc][,value][,MF=FS]
```

where:

rc is a self-defining term, or the location of a fullword return value to be loaded into GR15. If omitted, the return code is zero. rc may be expressed as a register number in parentheses.

value is a self-defining term, or the location of a fullword return value to be loaded into GR0. It may be expressed as a register number in parentheses.

MF=FS specifies that the save area pointed to by register 13 was obtained dynamically and is to be released. A SUP 1 is generated if the FREESPAC is unsuccessful.

Comments

This macro requires that the save area be properly linked on entry to the subroutine, as is done by the ENTER macro.

Examples

```
EXIT
EXIT 0
EXIT 12,15,MF=FS
EXIT ,RESULT,MF=FS
EXIT (4),(5)
```

CALL

Purpose

To cause control to be passed to a control section at a specified entry point.

Prototype

[label] CALL {epname|(15)} [,{parameter|(parameters)}] [,VL]

where:

epname is the name of the external entry point to be given control. If (15) is designated, register 15 must contain the address of the entry point.

parameter is a symbolic name.

parameters is one or more symbolic name(s) or null parameters.

VL specifies that the list of addresses generated is to contain a 1 in bit 0 of the last parameter.

Comments

Registers are not allowed as parameters, nor are the list and execute forms implemented. The convention followed is that register 1 contains an address that in turn points to a sequential list of address constants which point to the actual parameters. If parameters are omitted, register 1 will be unchanged and no address constants will be generated. Before the call is made, register 13 must point to the calling program's save area. Upon entry to the called subroutine, register 14 will contain the return address, and register 15 will contain the address of the entry point in the called subroutine.

The expanded code destroys the contents of registers 14 and 15. If VL and/or other parameters are given, the contents of register 1 are also destroyed.

The called program may change the contents of registers 0 and 1, and the condition code.

Examples

```
CALL  PROCESS
CALL  CONVERT,BUFFER
CALL  MYWRITE,(BUFFER,,LENGTH),VL
CALL  READ
CALL  (15)
```

SCARDS, READ

Purpose

To read a card.

Prototypes

```
[label]  SCARDS  {buffer,length} [,EXIT=exit]
[label]  READ  {unit,buffer,length} [,EXIT=exit]
```

where:

buffer is the location of the input buffer. It may be specified as a symbol or as a register enclosed in parentheses which contains the address of the buffer.

length is the address of a halfword where the length of the record read will be placed. It may be specified as a symbol, or a register name or number in parentheses. If a symbol, the length of the record will be placed in the specified halfword. If length specifies a register, the length will be placed there.

exit specifies the exit to be taken for a nonzero return code (end-of-file exit). It may be specified as a symbol or a register containing the exit routine address.

unit is the logical I/O unit to be read from. This parameter is ignored because ASSIST does not have this facility.

Comments

The contents of registers 0, 1, 14, and 15 are destroyed during the call. Register 13 must point to the calling program's save area.

The condition code will be changed.

Examples

```
SCARDS CARD,LENGTH,EXIT=EOF
SCARDS (5),(6)
SCARDS BUFFER,(GR5),EXIT=(4)
READ   4,BUFF,LEN,EXIT=(4)
```

SAVE

Purpose

The SAVE macro instruction causes the contents of the specified registers to be stored in the save area at the address contained in register 13. An entry point identifier can optionally be specified. The SAVE macro instruction should be written only at the entry point of a program because the code resulting from the macro expansion requires that register 15 contain the address of the save macro instruction.

Prototype

```
[label]  SAVE  (reg1[,reg2]),[T] [,id name]
```

where:

reg1,reg2 is the range of registers to be stored in the save area at the address contained in register 13. The registers should be designated so they are stored in the order 14, 15, and 0 through 12 when used in a STM instruction. The registers are stored in words 4 though 18 of the save area. If only one register is designated, only that register is saved.

T specifies that registers 14 and 15 are to be stored in words 4 and 5, respectively, of the save area. If both T and reg2 are designated, and reg1 is either 14, 15, 0, 1, or 2, all of the registers 14 through the reg2 value are saved.

id name is an identifier to be associated with the SAVE macro instruction. The name may be up to 70 characters and may be a complex name. If an asterisk is coded, the identifier is the symbol label associated with the SAVE macro instruction; if the name field is blank, the identifier is the control section name. If the CSECT instruction name field is blank, the operand is ignored.

Comments

This macro is provided for compatibility with the OS/360 SAVE macro. ENTER is recommended for MTS compatibility.

Examples

```
SAVE (14,12),T,*
SAVE (6)
SAVE (14),T,KRUNCH
```

RETURN

Purpose

The RETURN macro instruction is used to return control to the calling program and to signal normal termination of the returning program. The return of control is always made by executing a branch instruction using the address in register 14. The RETURN macro instruction can be written to restore a designated range of registers, to provide the proper return code in register 15, and to flag the save area used by the returning program.

Prototype

```
[label]  RETURN  [(reg1[,reg2])][,T][,RC=ret]
```

where:

reg1,reg2 is the range of registers to be restored from the save area pointed to by the address in register 13. The registers should be designated to cause the loading of registers 14, 15, and 0 through 12 when used in a LM instruction. If reg2 is not designated, only the register designated by reg1 is loaded. If the operand is omitted, the register contents are not altered.

T causes the control program to flag the save area used by the returning program. After the registers have been loaded, a byte of all ones (X'FF') is placed in the high-order byte of word four of the save area.

ret is the return code to be passed to the calling program. The return code should have a maximum value of 4095; it will be placed right-adjusted in register 15 before return is made. If RC=(15) is coded, it indicates that the return code has been previously loaded into register 15; in this case the contents of register 15 are not altered or loaded from the save area. (If this operand is omitted, the contents of register 15 is determined by the reg1, reg2 operands.)

Comments

This macro is provided for OS/360 compatibility. MTS users should use EXIT.

Examples

```
RETURN (14,12),T,RC=(15)
RETURN (6),,RC=8
RETURN (14,2),,RC=0
```

SPRINT, WRITE

Prototypes

```
[label]   SPRINT  {'text'|buffer,length} [,EXIT=exits]

[label]   WRITE  unit,{'text'|buffer,length}[,EXIT=exits]
```

where:

text is a literal message to be written.

buffer is the location of the output buffer. It may be specified as a symbol or as a register enclosed in parentheses which contains the address of the buffer.

length is the size of the output buffer in bytes. It may be specified as the name of a halfword containing the length, as a self-defining term, or as a register enclosed in parentheses which contains the length. Unlike what is allowed by the macro in *SYSMAC, it may not be omitted.

exits specify exits to be taken for nonzero return codes. Since no return codes are returned by the ASSIST write routines, this parameter is ignored.

unit is the logical I/O unit. This parameter is ignored since ASSIST does not have this facility.

Comments

The list and execute form of these macros presently available in *SYSMAC.

The contents of registers 0, 1, 14, and 15 are destroyed during the call. Register 13 must point to the calling program's save area.

The condition code will be changed.

Examples

```
SPRINT ' MESSAGE'
WRITE  6,BUFFER,121,EXIT=ERROR
WRITE  1,(4),(5),EXIT=(EOT,INVBSP,ERROR)
SPRINT (7),100
```

May 1983

ERROR

Purpose

To assemble a call to the ERROR subroutine.

Prototype

```
[label] ERROR
```

Comments

The ERROR system subroutine is called and the program terminates abnormally.

The contents of registers 14 and 15 are destroyed during the call. Register 13 must point to the calling program's save area.

The condition code will be changed.

Examples

```
BOMB     ERROR
```

DFIX, EFIX

Purpose

To convert a floating-point number (in a floating-point register) to an integer (in a general register).

Prototypes

```
[label] DFIX fpr,gr[,WA=wkarea]
[label] EFIX fpr,gr[,WA=wkarea]
```

where:

fpr is the floating-point register.

gr is the general register.

wkarea (optional) is a keyword parameter designating a doubleword-aligned work area of 16 bytes. If omitted, the macro will allocate an in-line work area.

Comments

DFIX converts a long-precision, floating-point number (8 bytes); EFIX converts a short-precision, floating-point number (first 4 bytes of a floating-point register). The contents of the specified floating-point register are restored at the end of the macro call. Note that it is possible to convert a floating-point number that is too big to fit (as an integer) into a general register, but the results will be meaningless since in order to make the floating-point number fit into a general register, the number is truncated. No attempt is made to signal this error.

After execution, the condition code will be set as follows:

```
0  if value=0
1  if value<0
2  if value>0
```

Example

```
LBL      DFIX  0,0
```

FLOAT

Purpose

To convert the contents of a general register or a fullword aligned area in storage into a floating-point number and leave the converted number in a floating-point register.

Prototype

[label] FLOAT arg1,arg2

where:

arg1 can either be a general register or a fullword-aligned location; if arg1 specifies a general register, the argument must be enclosed in parentheses. The contents of GR0 are destroyed if arg1 specifies a storage location.

arg2 is the floating-point register into which the results are placed.

Comments

Addressability of the literal pool is required.

The condition code will be changed.

Examples

```
LABEL    FLOAT (6),4

LABEL    FLOAT FULLWORD,0
```

FREESPAC

Purpose

To assemble a call to the FREESPAC subroutine.

Prototype

[label] FREESPAC [loc]

where

loc (optional) is the location of a fullword containing the address of a region allocated by a call to the GETSPACE subroutine, or the number of a register (in parentheses) which contains the address of such a region. If this parameter is omitted, it is assumed that the location is given in GR1.

Comments

This macro is not identical to the *SYSMAC version of the FREESPAC macro.

The contents of registers 1, 14, and 15 are destroyed during the call. Register 13 need not point to a save area.

The condition code will be changed.

Examples

```
.
.
.
LABEL    FREESPAC           , FREE UP THE GETSPACE REGION
.
.

.
.
LABEL    FREESPAC AREA        RELEASE OUR AREA
.
.
```

GETSPACE

Purpose

To assemble a call to the GETSPACE subroutine.

Prototype

[label] GETSPACE [length]

where:

length (optional) is a number or symbolic expression specifying the number of bytes of storage wanted. If omitted, the length is assumed to be in GR1.

Comments

The contents of registers 1, 14, and 15 are destroyed during the call. Register 13 need not point to a save area.

The condition code will be changed.

Examples

```
LABEL    GETSPACE 4096

LABEL    GETSPACE AMOUNT

LABEL    GETSPACE
```

REQU

Purpose

To generate EQU statements for the general registers.

Prototype

```
REQU
```

Comments

The code generated is:

```
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
RA       EQU   10
RB       EQU   11
RC       EQU   12
RD       EQU   13
RE       EQU   14
RF       EQU   15
```

SYSTEM

Purpose

To assemble a call to the SYSTEM subroutine.

Prototype

```
[label] SYSTEM
```

Comments

The SYSTEM macro produces the following code:

```
[LABEL]  L     15,=V(SYSTEM)
         BALR  14,15
```

Example

```
FINIS    SYSTEM
```

MTS 14: 360/370 Assemblers in MTS

May 1983

INDEX

Reader's Comment Form

360/370 Assemblers in MTS
Volume 14
May 1983

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or UNYN.

Date _____

Name _____

Address _____

_____

_____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Update Request Form

360/370 Assemblers in MTS
Volume 14
May 1983

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you desire to have these updates mailed to you, please submit this form.

Updates are also available in the memo files at the Computing Center, NUBS, and UNYN; there you may obtain any updates to this volume that may have been issued before the Computing Center receives your form. Please indicate below if you desire to have the Computing Center mail to you any previously issued updates.

Name _____

Address _____

_____

_____

Previous updates needed (if applicable):_____

The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or UNYN. Campus Mail addresses should be given for local users.

Publications
Computing Center
The University of Michigan
Ann Arbor, Michigan 48109

Users associated with other MTS installations (except the University of British Columbia) should return this form to their respective installations. Addresses are given on the reverse side.

Addresses of other MTS installations:

Publications Clerk
352 General Services Bldg.
Computing Services
The University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Information Officer, NUMAC
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne
England NE1 7RU

Rensselaer Polytechnic Institute
Documentation Librarian
310 Voorhees Computing Center
Troy, New York 12181

Simon Fraser University
Computing Centre
User Services Information Group
Burnaby, British Columbia
Canada V5A 1S6

Wayne State University
Computing Services Center
Academic Services Documentation Librarian
5950 Cass Ave.
Detroit, Michigan 48202