CA-MetaCOBOL™ +

Macro Facility Tutorial

Release 1.1

R203M+11DRP

Release 1.1, January, 1992
Updated March, 1992, May 1992

Contents

1. About This Manual

This tutorial describes how to write CA-MetaCOBOL+ macros. The most widely used facilities for the CA-MetaCOBOL+ macro processor are presented here; however, some details are omitted. The CA-MetaCOBOL+ *Macro Facility Reference* is the authoritative reference to the complete language.

The CA-MetaCOBOL+ *User Guide* also provides useful details.

1.1 Purpose

This is a training manual. It assumes that the reader has COBOL programming experience but no CA-MetaCOBOL+ experience.

Subjects are presented as building blocks, so that your understanding of the CA-MetaCOBOL+ macro language will progress step by step. For your convenience and review, each chapter ends with a summary of key points and an exercise that covers the learning to that point. Check your progress by reviewing the summaries and working through the exercises. The answers are available at the end of the book.

When you have completed this guide, you will be able to:

- Write CA-MetaCOBOL+ macros that analyze, reformat, and translate COBOL source programs.
- Build words and variables
- Perform condition test and control transfers within and among macro models.
- Acquire data items and their attributes while in the Procedure Division and generate code to one division while in another division.
- Parse and analyze input source of varied formats.
- Control the format of output listings and output files.
- Test and debug macros.

1.2 Organization

The Macro Facility Tutorial is organized as follows:

Chapter	Description
1	Discusses the purpose of the manual, gives a list of CA-MetaCOBOL+ documentation, and explains notation conventions for CA-MetaCOBOL+.
2	Gives an introduction to CA-MetaCOBOL+ macros and explains how words are translated.
3	Explains how to write basic CA-MetaCOBOL+ macros.
4	Gives procedures for writing string macros and describes interrelationships between macros.
5	Describes how to begin programming using the CA-MetaCOBOL+ Macro Language.
6	Discusses how to use variables with the CA-MetaCOBOL+ Macro Language.
7	Explains how to leave macros and how to define branches, conditions, and constructs.
8	Describes how to acquire symbolic-operand attributes and modify symbolic operands.
9	Explains the process of parsing input source.
10	Describes how to control the output of programs written with the CA-MetaCOBOL+ Macro Language.
11	Explains how to use event-dependent macros.
12	Gives standards and debugging techniques for macro writing.
Appendix A	Provides solutions to each exercise in this tutorial.
Appendix B	Illustrates the nesting of CA-MetaCOBOL+ macros.
Appendix C .	Lists the attributes of symbolic operands
Appendix D	Lists the values returned by the NOTE register.
Glossary	Explains standard terms used with CA-MetaCOBOL+.

1.3 Publications

In addition to this manual, the following publications are supplied with CA-MetaCOBOL+.

Title	Contents
Introduction to CA-MetaCOBOL+	Introduces the CA-MetaCOBOL+ Work Bench, Structured Programming Facility, Quality Assurance Facility, CA-DATACOM/DB Facility, Macro Facility, Panel Definition Facility, and the Online Programming Language.
Installation Guide - MVS	Explains how to install CA-MetaCOBOL+ in the MVS environment.
CA-ACTIVATOR Installation Supplement – MVS	Explains how to install CA-MetaCOBOL+ in the MVS environment using CA-ACTIVATOR.
Installation Guide - VSE	Explains how to install CA-MetaCOBOL+ in the VSE environment.
Installation Guide - CMS	Explains how to install CA-MetaCOBOL+ in the VM environment.
User Guide	Explains how to customize, get started, and use CA-MetaCOBOL+. Includes information on keyword expansion, the CA-MetaCOBOL+ translator, and CA macro sets and programs.
Structured Programming Guide	Introduces the Structured Programming Facility. Includes information on creating, testing, and maintaining structured programs.
Macro Facility Reference	Includes detailed information on the program flow of the CA-MetaCOBOL+ macro translator, macro format, definition of comments, macro nesting, macro prototypes, symbolic words, and model programming.

Title	Contents
Quality Assurance Guide	Introduces the Quality Assurance Facility. Includes all the necessary information to perform quality assurance testing on COBOL source programs.
Program Development Guide CA-DATACOM/DB	Includes all the information necessary to develop programs that make full use of the functions and features of the CA-DATACOB/DB environment.
Program Development Reference CA-DATACOM/DB	Contains all CA-DATACOM/DB Facility constructs and statements.
Panel Definition Facility Command Reference	Contains all Panel Defintion Facility commands.
Panel Definition Facility User Guide	Includes all the information necessary to create, edit, duplicate, rename, delete, index, and print panel definitions and members. Also describes how to generate BMS source.
Online Programming Language Reference	Contains all Online Programming Language statements.
Online Programming Language Guide	Provides further instruction for using Online Programming Language statements.
PC User Guide	Explains how to use CA-MetaCOBOL+/PC. Includes information on the CA-MetaCOBOL+ translator and CA macro sets and programs. Also decribes the relationship between CA-MetaCOBOL+ and CA-MetaCOBOL+/PC.
Program Development Guide CA-DATACOM/PC	Describes how to develop programs that use the CA-DATACOM/PC environment.
String Manipulation Language Guide	Introduces the String Manipulation Language, which provides string handling and inspection capabilities unavailable in COBOL.

All manuals are updated as required. Instructions accompany each update package.

1.4 Notation Conventions

The following conventions are used in the command formats throughout this manual:

UPPERCASE BOLD	is used to display commands or keywords you must code exactly as shown.
lowercase italic	is used to display information you must supply. For example, DASD space parameters may appear as *xxxxxxx xxxxxxx xxxxxxx*.
Underscores	either show a default value in a screen image, or represents the highlighting of a word in a screen image.
Brackets [ ]	mean that you can select one of the items enclosed by the brackets; none of the enclosed items is required.
Braces { }	mean that you must select one of the items enclosed by the braces.
Vertical Bar |	separates options. One vertical bar separates two options, two vertical bars separate three options, and so on. You must select one of the options.
Ellipsis . . .	means that you can repeat the word or clause that immediately precedes the ellipsis.

1.5 Summary of Revisions

The following modifications and enhancements have been made to this manual for Version 1.1 of MetaCOBOL+.

- Minor technical corrections and editorial changes have been made to macro set examples, JCL samples, and other text throughout the manual.
- Two symbolic operands, &n(R) and &n(R,L), have been added to the chart of symbolic operands in Section 4.1, "String Macro Prototype."

- In Appendix C, "Symbolic Operand Attributes," three attributes have been added: EXTERNAL, GLOBAL, and NAME CHECK.

- In Appendix D, "&SETR Note Register," three values have been added: B (Blank Lines), C (CICS Statements), and Q (SQL statements).

2. Getting Started - Introduction to CA-MetaCOBOL+ Macros

This chapter introduces the CA-MetaCOBOL+ Translator, macro processing, and CA-MetaCOBOL+ macros. Understanding the concepts presented here is essential to writing effective macros.

2.1 What Is the CA-MetaCOBOL+ Translator?

The CA-MetaCOBOL+ Translator is a member of a family of programs known as *language processors.* This family includes assemblers, compilers, interpreters, and the like. All of these programs share the ability to translate computer language statements into a different form of the same language or into an entirely different language.

Specifically, the Translator is a COBOL-oriented macro processor. Traditionally, COBOL preprocessors have been highly specialized - designed to handle specific tasks. CA-MetaCOBOL+'s Translator is a general-purpose preprocessor that can be customized by means of its macro language to solve a broad range of problems.

2.2 What Does the Translator Do?

The CA-MetaCOBOL+ Translator performs a range of useful functions for the COBOL programmer.

The Translator automatically formats COBOL programs based on options selected by each installation and applies translation rules laid out in CA-MetaCOBOL+ macros.

When installed with the macro procedure sets distributed by CA, the Translator can provide a high-level COBOL-like language that meets programming needs not currently handled in COBOL, a COBOL program conversion facility, and a short-form facility.

In addition, the Translator allows each installation to define its own translation rules. These *macro definitions* can be used along with the CA macro procedure sets to customize the translation process. The CA-MetaCOBOL+ macro programmer can define the source language that COBOL programmers use and the translation rules that process these statements. This guide describes how to write such macros.

The ability of the Translator to expand high-level, COBOL-like input into compilable COBOL source is demonstrated by the following sample. The code at the top is the input source. A series of macros, not shown here, translate that code into the COBOL report program that follows.

Input Source

```
$ID -ID ALPHA BY MAGEE
    -INS CA, INC PRINCETON, NJ.  -DC
$ED -CS -IOFC
    -SEL INPUT-FILE TO UT-S-INPUT.
$DD -FS
FD   INPUT-FILE -LRS -RMF -RC 80 -C -BC 0 -REC.
01   INPUT-RECORD.
     INPUT-RECORD.
     02 INPUT-DATA  P=X(36).
     02 FILLER  P=X(44).
-WS
77   LEFT-LINE-NUMBER P = 99 VALUE 01.
77   RIGHT-LINE-NUMBER P = 99 VALUE 01.
$PD
     $INPUT INPUT-FILE.
     $PRINT REPORT-FILE FOR 49 LINES.
     $HEAD
     4 `1...5...10...15....20...25...30...35.'
     50 `PAGE' 55 $PAGE.
     $DETAIL  1 P=99 LEFT-LINE-NUMBER
              3 `.' 4 INPUT-DATA
              40 `.' 41 P=99 RIGHT-LINE-NUMBER
     IF LEFT-LINE-NUMBER = 48
       MOVE 01 TO LEFT-LINE-NUMBER
                  RIGHT-LINE-NUMBER
     ELSE
       ADD 1 TO LEFT-LINE-NUMBER
                RIGHT-LINE-NUMBER
```

Translated Output:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    ALPHA.
AUTHOR.
    MAGEE
INSTALLATION.
    CA INC,  PRINCETON, NJ.
DATE-COMPILED.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECTION INPUT-FILE
                       ASSIGN TO UT-S-INPUT.
    SELECT REPORT-FILE
                       ASSIGN TO UT-S-REPORT.
```

```
DATA DIVISION.
FILE SECTION.
FD INPUT-FILE
                       LABEL RECORD STANDARD
                       RECORDING MODE F
                       RECORD CONTAINS 80 CHARACTERS
                       BLOCK CONTAINS 0 RECORDS.
01  INPUT-RECORD.
    02 INPUT-DATA      PICTURE IS X(36).
    02 FILLER          PICTURE IS X(44).
FD  REPORT-FILE
                       RECORDING MODE F
                       LABEL RECORDS OMITTED
                       DATA RECORD PRINT-LINE.
01  PRINT-LINE         PIC X(133).
WORKING-STORAGE SECTION.
77  LEFT-LINE-NUMBER   PICTURE IS 99 VALUE 01.
77  RIGHT-LINE-NUMBER  PICTURE IS 99 VALUE 01.
01  ZZ-REPORT-ITEMS-ZZ SYNC.
    02 ZZ-LEVEL-ZZ     PIC S99 COMP.
    02 ZZ-LEV-ZZ       PIC S99 COMP.
    02 ZZ-SUBT-ZZ      PIC S99 COMP.
    02 ZZ-USER-GATE-ZZ PIC X.
    02 ZZ-LR-ZZ        PIC X VALUE `F`.
    02 ZZ-REPORT-HOLD-ZZ PIC X VALUE SPACE.
    02 ZZ-REPORT-ZZ    PIC X.
    02 ZZ-SPACE-ZZ     PIC X VALUE `1'.
    02 ZZ-SP-ZZ        PIC S9 COMP-3 VALUE +0.
    02 ZZ-PCTR-ZZ      PIC S9(5) COMP-3 VALUE +0.
    02 ZZ-LCTR-ZZ      PIC S999 COMP-3 VALUE +0.
    02 ZZ-OFLO-SW-ZZ   PIC X VALUE SPACE.
01  HEADER-1
    02 FILLER          PIC X(4) VALUE SPACE.
    02 FILLER          PICTURE X(37) VALUE
        `1...5...10...15..CC20...25...30...35.'.
    02 FILLER          PIC X(9) VALUE SPACES.
    02 FILLER          PICTURE X(4) VALUE `PAGE'.
    02 FILLER          PIC X(1) VALUE SPACES.
    02 HEADER-1-PAGE   PIC Z(5).
01  Z-DETAIL.
    02 FILLER          PIC X(1) VALUE SPACES.
    02 Z-DETAIL-FIELD-1 PIC 99.
    02 FILLER          PICTURE X(1) VALUE `.'.
    02 Z-DETAIL-FIELD-2 PIC X(36).
    02 FILLER          PICTURE X(1) VALUE `.'.
    02 Z-DETAIL-FIELD-3 PIC 99.
PROCEDURE DIVISION.
10-ROOT SECTION 1.
    OPEN INPUT INPUT-FILE.
    GO TO 30-OPEN.
```

```
20-READ.
    IF ZZ-LR-ZZ = `N'
      CLOSE INPUT-FILE
      GO TO 40-CLOSE.
    READ INPUT-FILE
      AT END
      MOVE `N' TO ZZ-LR-ZZ.
    IF ZZ-LR-ZZ = `N'
      GO TO 700-FINAL
    GO TO 300-PRINT.
30-OPEN.
    OPEN OUTPUT REPORT-FILE.
    GO TO 20-READ.
40-CLOSE.
    CLOSE REPORT-FILE.
    GOBACK.
50-EXIT.
    EXIT.
200-HEAD.
    MOVE +0 TO ZZ-LCTR-ZZ.
    MOVE `1' TO ZZ-SPACE-ZZ.
    MOVE +1 TO ZZ-SP-ZZ.
    ADD+1 TO ZZ-PCTR-ZZ.
    MOVE ZZ-PCTR-ZZ TO HEADER-1-PAGE
    WRITE PRINT-LINE FROM HEADER-1 AFTER POSITIONING
      ZZ-SPACE-ZZ LINES.
    ADD ZZ-SP-ZZ TO ZZ-LCTR-ZZ.
    MOVE SPACE TO ZZ-SPACE-ZZ
    MOVE +1 TO ZZ-SP-ZZ.
290-HEAD-EXIT.
    EXIT.
300-PRINT
    IF ZZ-LR-ZZ = `F'
      PERFORM 200-HEAD THRU 290-HEAD-EXIT
      MOVE `M' TO ZZ-LR-ZZ.
    IF ZZ-LR-ZZ = `N'
      GO TO 20-READ.
    IF ZZ-OFLO-SW-ZZ = `T'
      MOVE SPACE TO ZZ-OFLO-SW-ZZ
      PERFORM 200-HEAD THRU 290-HEAD-EXIT.
      MOVE LEFT-LINE-NUMBER TO Z-DETAIL-FIELD-1.
      MOVE INPUT-DATA TO Z-DETAIL-FIELD-2.
      MOVE RIGHT-LINE-NUMBER TO Z-DETAIL-FIELD-3.
    IF LEFT-LINE-NUMBER = 48
      MOVE 01 TO LEFT-LINE-NUMBER RIGHT-LINE-NUMBER
    ELSE
      ADD 1 TO LEFT-LINE NUMBER RIGHT-LINE-NUMBER.
    IF ZZ-LCTR-ZZ > +48
      PERFORM 200-HEAD THRU 290-HEAD-EXIT.
    WRITE PRINT-LINE FROM Z-DETAIL AFTER POSITIONING
```

```
               ZZ-SPACE-ZZ.
             ADD ZZ-SP-ZZ TO ZZ-LCTR-ZZ.
             MOVE SPACE TO ZZ-SPACE-ZZ
             MOVE +1 TO ZZ-SP-ZZ.
         400-PROCESS.
             GO TO 20-READ.
         700-FINAL.
             GO TO 20 READ.
```

2.3 What Is a Macro?

A macro is a facility for replacing one sequence of symbols by another.

P.J. Brown, *Macro Processors*
(London: John Wiley & Sons,1974), p.4

The CA-MetaCOBOL+ macro, supported by the CA-MetaCOBOL+ Translator, defines how to replace a word or sequence of words of COBOL-like input source by a word or sequence of words of compilable COBOL output source. Translation of each input source program is accomplished by the Translator under the control of a number of CA-MetaCOBOL+ macros, each of which acts upon one series of input words.

A macro is like a subroutine: a short but explicit "call" in the input source is processed as specified by the macro and expanded into syntactically correct output source. The difference between the two is important to understand, though. The CA-MetaCOBOL+ macro is applied before the output source is compiled; it produces compilable code. The COBOL subroutine is applied after compilation, when the program is run.

A macro consists of two parts:

- The macro *prototype* defines what words to remove from the input source.
- The macro *model* defines what words replace the prototype.

The prototype and the model are the essential tools of the macro writer, and these will be discussed at length shortly.

2.4 Translation Overview

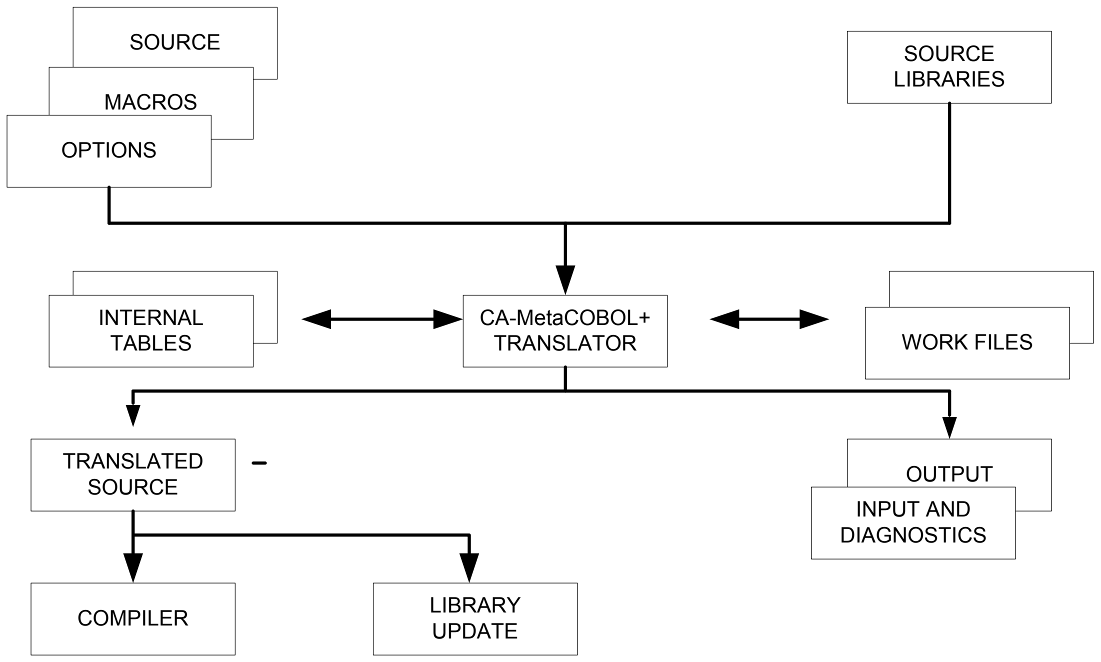
The following diagram presents, at a high level, the flow of a COBOL program and CA-MetaCOBOL+ macros when the program is passed through the Translator.

Translator input consists of CA-MetaCOBOL+ *option statements,* CA-MetaCOBOL+ *macros*, and a single *source program.* Macros and the source program can also be included from standard source libraries, CA-LIBRARIAN, or CA-PANVALET. Internal *tables* and *work files* are used to store the macros after they are read and to use later during the translation process.

The output consists of two parts:

1. a COBOL *source file,* which contains the translated source program and can be directed to the compiler or returned to the source librarian.
2. a variety of optional *listings*.

The following diagram provides an overview of the CA-MetaCOBOL+ Translator:



The macro translation process works as follows:

1. During Initialization, the Translator reads options and macros in one input stream. Options come from Option statements and/or the EXEC statement PARM field under OS.

2. The macros are read, interpreted, and stored in internal tables.

3. The first COBOL division header terminates macro loading and initiates the translation phase.

4. The Translator examines the source program word-for-word looking for a word, a pattern of words, or an event that exactly matches a macro prototype in the internal tables.

5. Unmatched source words are passed to the output unchanged.

6. When a match occurs, any input source words that match the prototype are removed from the input and the action specified in the associated macro model is performed.

   For example, a macro definition could place the same word back in the source output and issue a diagnostic, substitute another word, or generate many lines of COBOL logic.

   If the result of a substitution matches another macro prototype, the action specified in the second macro model is performed. The Translator does not, however, match substituted words against the prototype of the macro from which they originated. Substituted words are passed to the output when they do not match another prototype.

7. Source output is formatted according to the conventions selected when CA-MetaCOBOL+ was installed. This format can be overridden at translation time.

8. The Translator prepares output listings. The two most frequently used are the Input and Diagnostics listing, containing macro definitions and the input source program, and the Output listing, containing the translated source program.

   The Output listing is always produced, but you can control production of the Input and Diagnostics listing using CA-MetaCOBOL+ Options.

The CA-MetaCOBOL+ *User Guide* contains a complete description of the Translator's input and output, including all Option statements.

2.5 How Words Are Translated

Because COBOL, like English, is comprised of words separated by spaces, the CA-MetaCOBOL+ Translator proceeds through the input source one word at time. It compares each word against the tables of stored macro prototypes. The Translator can recognize basic COBOL structures such as level numbers, Area A words, and COBOL verbs and can produce formatted COBOL source. However, it must be instructed, by means of the macro, exactly what to look for in the source program and what to do with each word that it acquires.

As complex and sophisticated as macro translations can become, they begin by evaluating the specific words that make up the COBOL statements and COBOL data and procedure references.

The following figure illustrates each step in the processing a sample macro.

	Macros	
Input Source	**Prototype : Model**	**Output Source**
	PIC : PICTURE	
02 PLANT PIC X(5).	VAL : VALUE	
	RDF : REDEFINE	
	PIC : PICTURE	
PLANT PIC X(5).	VAL : VALUE	02
	RDF : REDEFINE	
	PIC : PICTURE	
PIC X(5).	VAL : PICTURE	02 PLANT
	RDF : REDEFINE	
	PIC : PICTURE	
PICTURE X(5).	VAL : VALUE	02 PLANT
	RDF : REDEFINE	
	PIC : PICTURE	
X(5).	VAL : VALUE	02 PLANT PICTURE
	RDF : REDEFINE	
	PIC : PICTURE	
	VAL : VALUE	02 PLANT PICTURE X(5)
	RDF : REDEFINE	
	PIC : PICTURE	
	VAL : VALUE	02 PLANT PICTURE X(5).
	RDF : REDEFINE	

Three macros have been loaded. They specify the following rules:

Prototype Removes	**Model Substitutes**
PIC	PICTURE
VAL	VALUE
RDF	REDEFINE

The following process is shown:

1. The input source consists of a COBOL data definition for the item PLANT, including five words: a level number, a data-name, the word PIC, a COBOL picture character-string, and a period (.).

2. The level number fails to match any prototype. It is placed unchanged in the output source.

3. PLANT fails to match any prototype and is placed unchanged in the output source.

4. PIC matches a prototype. It is removed from the input source and replaced by PICTURE.

5. PICTURE fails to match any prototype and is placed unchanged in the output source.

6. The picture character-string fails to match any prototypes and is place unchanged in the output source.

7. The period fails to match any prototypes and is placed unchanged in the output source.

As a result, the abbreviation PIC in the input source is replaced in the output source by the word PICTURE. The VAL and RDF macros have no effect on the translation, because no words in the input source and no words replaced by other macros match their prototypes.

Note that the Translator accommodates the word PICTURE, which is four characters longer than the input word PIC. It will automatically adjust the output source to accommodate the replacement or removal of words in the source input.

2.6 What Is a CA-MetaCOBOL+ Word?

In the preceding example, the terminating period is treated as a separate word. To the Translator, a word is a string of characters fitting one of the following descriptions:

- Numeric literal
- Non-numeric literal
- Terminating punctuation character.
- Contiguous character string not exceeding 30 characters bounded by a space, margin, or terminating punctuation character. The string may not contain an embedded space, quotation mark, or punctuation character. Such words can be a verb, reserved word, data-name, procedure-name, etc.

2.7 Reserved System Abbreviations

The CA-MetaCOBOL+ Translator provides special abbreviations which are reserved to represent the COBOL division headers. These abbreviations must begin in Area A of the input source. No other abbreviations may be used for division headers. The standard COBOL division headers may be coded in full.

CA-MetaCOBOL+ Abbreviation	**Translation**
$ID	Identification Division.
$ED	Environment Division.
$DD	Data Division.
$PD	Procedure Division.

2.8 MVS/VSE Translation Execution JCL

To take full advantage of this guide you should work the exercises and build your own macros.The following JCL is supplied as a guide for executing CA-MetaCOBOL+ in an MVS environment. You may want to alter the SYSOUT class and BLKSIZE (in multiples of sizes shown), SPACE, and UNIT.

Note: The JCL below is sample start-up JCL only. You must supply the *options*, *macros*, and *Source Program* indicated in the last three lines of the sample below. Refer to the CA-MetaCOBOL+ *User Guide* for more information. If you are using CA-MetaCOBOL+/PC, refer to the CA-MetaCOBOL+/PC *User Guide* for equivalent SET statements.

```
 //[stepname] EXEC PGM=METACOB[,PARM=`options']
[//STEPLIB    DD   DSN-user.loadlib,DISP=SHR]
 //LSTIN      DD   SYSOUT=A[,DCB=BLKSIZE=121]
 //PUNCHF     DD   SYSOUT=B[,DCB=BLKSIZE=80]
[//AUX        DD   SYSOUT=B[,DCB=BLKSIZE=80]]
 //FE         DD
UNIT=SYSDA,SPACE=(TRK,(20,10))[,DCB=BLKSIZE=148]
 //FM         DD
UNIT=SYSDA,SPACE=(TRK,(20,10))[,DCB=BLKSIZE=148]
 //CARDF      DD   *[,DCB=BLKSIZE=80]
 [OPTION options]
 [macros]
  Source Program
/*
```

The job stream outlined below is supplied as a guide for executing CA-MetaCOBOL+ in a VSE environment. You may use standard labels for the disk work areas.

Note: The JCL below is sample start-up JCL only. You must supply the *options*, *macros*, and *Source Program* indicated in the last three lines of Segments 1 and 2 below. Refer to the CA-MetaCOBOL+ *User Guide* for more information. If you are using CA-MetaCOBOL+/PC, refer to the CA-MetaCOBOL+/PC *User Guide* for equivalent SET statements.

To send output to a disk, use Segment 1. To send output to a tape, use Segment 2.

Segment 1

To `punch' CA-MetaCOBOL+ output to a disk, complete the JCL as shown below.

```
 // DLBL IJSYS06,...
 // EXTENT SYS006
 // ASSGN SYS006,DISK,VOL=XXXXXX,SHR


  [// ASSGN SYS007,X`cuu']                  Auxiliary output.
   // EXEC METACOB

   [OPTION options]
   [macros]
    Source Program
   /*
```

Segment 2

To `punch' CA-MetaCOBOL+ output to an unlabeled tape, complete the JCL as shown below.

```
 // ASSGN SYS006,X`cuu'
    MTC REW,SYS006
    MTC WTM,SYS006

 [// ASSGN SYS007,X`cuu']                   Auxiliary output.
   // EXEC METACOB

   [OPTION options]
   [macros]
    Source Program
   /*
```

The JCL shown above is for getting started. For complete descriptions of MVS and VSE JCL, as well as all run-time options, see the *CA-MetaCOBOL+ User Guide*.

2.9 Summary - Introduction to CA-MetaCOBOL+ Macros

Listed below are the key points discussed in this section.

- The CA-MetaCOBOL+ Translator is a language processor that translates COBOL or COBOL-like input source into COBOL output source under the control of CA-MetaCOBOL+ macros.
- The macro programmer can define translation rules in CA-MetaCOBOL+ macros that define the input source language and customize translation.
- A number of macros may be used to translate a single source program. Each macro defines how to substitute a word or sequence of words in the input source by word(s) in the output source.
- A macro prototype defines what words to remove from the input source or an event in the input source.
- A macro model defines what words to substitute in the output source for the prototype or what other actions to take.
- Each word of source text is matched against macro prototypes independently of all other words. Each word of substituted text is also matched against the macro prototypes and is replaced with the model if found.

  In a sense, the only words output by the macro processor are words that do not match any macro prototypes.
- The Translator reads macros and source as input; macros are stored in internal tables; the first division header ends macro loading.
- Abbreviations reserved for COBOL division headers are $ID, $ED, $DD, $PD.

3. Getting Started - Writing Basic Macros

The first step in writing CA-MetaCOBOL+ macros is to understand the format of all macros and how to write the two simplest macro types-the Word macro and the Prefix macro.

3.1 Macro Format

A CA-MetaCOBOL+ macro comprises the following components:

Format:

```
type [division-code...  ] prototype : model
```

Type

The type is coded in the continuation column (cc. 7). The possible codes and their meanings are:

W Word Macro (searches for a single word).

P Prefix Macro (searches for a single word consisting of a constant prefix, or a first few characters of a word, and the variable suffix, or remainder of the word).

S String Macro (searches for one or more words, some of which are constant, others of which may be variable).

V or **U**
Verb Macro or Unverb Macro (V defines a word as reserved verb, U releases a previously defined verb; there is no model, see the *Macro Facility Reference.*

Division Code

The division coded in Area A (cc. 8-11).

The macro can contain any combination of these division codes: I (Identification Division), E (Environment Division), D (Data Division), and P (Procedure Division). These codes designate the COBOL divisions in which the macro is effective. Please note:

- If no code is specified, all divisions are assumed.
- Multiple division codes must be adjacent.
- The last division code must be followed by a space.

Prototype

The prototype is coded in Area B (cc. 12-72).

Each macro must contain a prototype defining what words to remove from the input source. The appearance of the prototype in the input source invokes the associated macro model. Please note:

- At least one blank must precede the macro prototype.
- A prototype may be extended over consecutive lines.

The prototype can be one word or a string of words, depending on the type code. The first word (or only word) of the macro prototype is the *macro name*. It can be a COBOL keyword, data-name, or procedure-name. It can also be a specially defined word such as an abbreviation. Other words specified within a prototype may be the same type as the macro name, or they can be variable words. (Variable words are called *symbolic operands*. More will be said about these later.)

Separator

The separator (:) is coded in Area B (cc. 12-72).

A colon (:) separates the prototype from the macro model. A space must immediately precede and follow the colon.

Model

The macro model is coded using margins A (cc. 8-11) and B (cc. 12-72) as in a COBOL program.

The model, specifying the words to replace the prototype or other actions to perform, follows the separator. The macro model can take one or more of the following forms.

- The model is omitted. All input words that call the macro are dropped from the COBOL output.
- The model contains COBOL words and statements. These words appear in the COBOL output in place of all input words that call the macro.
- The model contains skeleton COBOL statements. Words in the input source are substituted in the model to complete the skeleton statements and appear in the COBOL output in place of all input words that call the macro.
- The model contains CA-MetaCOBOL+ Translator directives. These directives control the placement and content of the COBOL output but do not appear in the output.

Embedded Comments

Any lines within a macro definition can be terminated by the characters:

```
/*
```

The remainder of the line can then contain comments. Standard COBOL comments (designated by an * in column 7) can also be used in the macro definitions.

Macro Definition Example

To illustrate how to define a macro, assume that a macro is required to replace every occurrence of the word JOE in a source program's Data and Procedure divisions with the word MOE.

1. Since the data-name to be replaced is a single word, a Word macro can be used. A type code of W is placed in column 7 of the first line of the macro definition.
2. The macro is to be effective in the Data and Procedure Divisions. The division codes D and P are placed following the type code (i.e., columns 8-9).
3. The word in the source program that is to trigger this macro is JOE. JOE, therefore, becomes both the macro prototype and the macro name (starting in column 12).
4. A colon separates the macro prototype from the macro model.
5. When JOE is found in the source program, it is to be replaced by the single word MOE. MOE is, therefore, the macro model.

The resulting macro is shown here with coding columns:

```
1...5...10...15...20...25...30...35...40...45...50

      WDP JOE  :  MOE     /* REPLACE JOE WITH MOE
```

Note that an embedded comment follows the macro definition.

The following excerpt from CA-MetaCOBOL+ Input includes the Word macro JOE and input source starting with $ID. The macro causes translation to the COBOL program shown as CA-MetaCOBOL+ Output. Note the automatic substitution of the expanded division headers (ENVIRONMENT DIVISION. for $ED, etc.). note also that JOE remains JOE in the AUTHOR statement in the Identification Division, but is changed to MOE in the Data and Procedure Divisions.

CA-MetaCOBOL+ Input:

```
WDP   JOE  :  MOE     /* REPLACE JOE WITH MOE
 $ID
 PROGRAM-ID.   DEMO.
 AUTHOR.   JOE.
 $ED
 .  .  .
 $DD
 WORKING-STORAGE SECTION.
 77 JOE PICTURE X.
 .  .  .
 $PD
    MOVE `A' TO JOE.
```

CA-MetaCOBOL+ Output

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   DEMO.
AUTHOR.   JOE.
ENVIRONMENT DIVISION.
.  .  .
DATA DIVISION.
WORKING-STORAGE SECTION.
77 MOE PICTURE X.
.  .  .
PROCEDURE DIVISION.
     MOVE `A' TO MOE.
```

3.2 Writing Word Macros

Word macros instruct the CA-MetaCOBOL+ Translator to look for a single word in the source. They are normally simple because they involve no variable information, but rather associate a single word (the prototype) with one or more words (the model).

W[*division-code...*] *prototype* : *model*

W

Code the W in the continuation column (cc. 7).

division-codes

Code division-codes in columns 8-11 to specify that the macro applies to one or more COBOL divisions and not to the entire program.

prototype and **model**

All that remains is to equate the abbreviation (prototype) to the COBOL text to be substituted (model). For example,

```
WP BUMP : ADD +1 TO
```

defines a short form-BUMP-that is expanded to ADD +1 TO when it is encountered in the Procedure Division.

Uses for Word Macros

- To define short forms of standard statements and of reserved COBOL words and clauses. For example, DO for the word PERFORM and FLAG for MOVE `S' TO ERROR-SWITCH.
- To define short forms of frequently coded data-names and procedure-names. For example, DNO can be coded in place for DEPARTMENT-NUMBER and then expanded in the output.
- To expand abbreviations of standard COBOL and of procedure-names automatically to full spellings. For example, PIC, COMP, and < can be translated to PICTURE, COMPUTATIONAL, and LESS THAN, respectively; and flow chart procedure-names such as P1 and P2, can be translated to descriptors BEGIN-PROGRAM and COUNT-TRANSACTIONS.
- To search for a word that is to be removed, flagged with a diagnostic, or converted to an alternate format.
- To define a macro subroutine. A macro subroutine is a macro whose model can be called from another macro model by specifying the macro name.

Short Form Examples

There are a number of terms used frequently in COBOL (such as FILLER, HIGH-VALUE, LOW-VALUE, PERFORM, GREATER THAN, and LESS THAN) which can be abbreviated to save coding. The abbreviations could be: -F, -HV, -LV, DO, >, and < respectively.

The following word macros define each of these short forms and use them in a COBOL excerpt. The resulting output follows.

CA-MetaCOBOL+ Input:

```
WD     -F    :    FILLER
SDP    -HV   :    HIGH-VALUE
WDP    -LV   :    LOW-VALUE
WP     DO    :    PERFORM
WP     >     :    GREATER THAN
WP     <     :    LESS THAN
$ID
 .  .  .
 $ED
 .  .  .
 $DD
 .  .  .
 WORKING-STORAGE SECTION.
 01    GROUP-ITEM.
       02 -F PIC X(20) VALUE -LV.
 .  .  .
 $PD
 .  .  .
       IF FIELDA < +1
         OR FIELD B> +9
         MOVE -HV TO GROUP-ITEM
         DO ERROR-ROUTINE.
```

CA-MetaCOBOL+ Output:

```
IDENTIFICATION DIVISION.
.  .  .
ENVIRONMENT DIVISION.
.  .  .
DATA DIVISION.
.  .  .
WORKING-STORAGE SECTION.
02     GROUP-ITEM.
       02 FILLER         PIC X(20) VALUE LOW-VALUE.
.  .  .
PROCEDURE DIVISION.
.  .  .
       IF FIELDA LESS THAN +1
          OR FIELDB GREATER THAN +9
          MOVE HIGH-VALUE TO GROUP-ITEM
          PERFORM ERROR-ROUTINE.
```

Notice the expansion of -F, and -LV in the Data Division, and >, <, -HV and DO in the Procedure Division. -HV and -LV apply to both the Data and Procedure Divisions.

Notice also that the greater-than and less-than symbols are translated into more than one word.

Exercise 1: Define short forms to generate the data-names BALANCE-ON-HAND, ISSUES-TODAY, RECEIPTS-TODAY, and PREVIOUS-BALANCE, and the paragraph-name COMPUTE-NEW-BALANCE.

For exercise solutions, see Appendix A.

3.3 Writing Prefix Macros

Many COBOL phrases and clauses pass a single variable to the COBOL compiler. Instead of having to write these out, it would be much easier for the COBOL programmer to write, for examples, BCR=10 instead of BLOCK CONTAINS 10 RECORDS or O=50 instead of OCCURS 50 TIMES. Specifying such abbreviations is a function of the Prefix macro.

Format:

P[~~division-code~~ ...] *prototype* : *model*

P

Code the P in the continuation column (cc. 7).

division-codes

Code the division codes in columns 8-11 to specify that the macro applies to one or more COBOL divisions and not to the entire program.

prototype

The macro prototype is actually the first few characters (the prefix) of a word. When the prefix is encountered in the input source, the remaining characters of the input source word are saved as a variable source word (the suffix).

For clarity, a special character such as an equal sign (=) is often used as the last character of the prefix. Optionally, the suffix may be represented in the prototype as an ampersand (&). The prototype then looks like the following sample:

```
BCR=&
```

model

The model of a Prefix macro usually shows expanded COBOL source with the place for the variable data reserved by an ampersand. For example:

```
...   : BLOCK CONTAINS & RECORDS
```

If the word BCR=10 is found in the input source, BCR= is the prefix and 10 is the suffix. The suffix from the input (10) is substituted for &, and the entire model (BLOCK CONTAINS 10 RECORDS) is output in place of the prefixed source word (BCR=10).

The ampersand, when used to carry variable data, is called a *symbolic operand*. Please note:

- Source words which call prefix macros cannot contain more than 30 characters (including prefix and suffix).
- The suffix cannot be specified in the input source as an alphanumeric literal. (For example, BCR=10 is legal; BCR=`10' is not.)

Uses for Prefix Macros

- To define short forms of reserved COBOL words and clauses and standard statements with one variable.
- To translate prefixed data-names and procedure-names to more descriptive names or qualified entries. For example, any input words prefixed with T- can be translated to words prefixed by TRANSACTION-, or words prefixed with T= can be qualified as OF TRANSACTION.
- To define a macro subroutine that passes a single variable.

Short Form with One Variable Example

The following Prefix macros define short forms for required COBOL clauses containing a single variable. The input source uses these short forms.

CA-MetaCOBOL+ Input

```
PD     BCR=&   : BLOCK CONTAINS & RECORDS
PD     O=&     : OCCURS & TIMES
PP     SKIP=&  : BEFORE ADVANCING & LINES
PP     M=&     : & OF MASTER-RECORD
  .  .  .
 $DD
 FILE SECTION.
 FD    INPUT-FILE BCR=10.
 01    INPUT-FILE-RECORD.
       02 INPUT-TABLE PIC X(15) O=20.
 WORKING-STORAGE SECTION.
 01    MASTER-RECORD.
       02 BALANCE PIC S9(5) COMP.
 $PD
       MOVE ZERO TO M=BALANCE.
       WRITE PRINT-LINE SKIP=2.
```

CA-MetaCOBOL+ Output:

```
             .   .   .
             DATA DIVISION.
             FILE SECTION.
             FD     INPUT-FILE BLOCK CONTAINS 10 RECORDS.
             01     INPUT-FILE-RECORD.
                    02 INPUT-TABLE PIC X(15) OCCURS 20 TIMES.
             WORKING-STORAGE SECTION.
             01     MASTER-RECORD.
                    02 BALANCE   PIC S9(5) COMP.
             PROCEDURE DIVISION.
                    MOVE ZERO TO BALANCE OF MASTER-RECORD.
                    WRITE PRINT-LINE BEFORE ADVANCING 2 LINES.
```

Use of the special character can also avoid oversights where unexpected words call Prefix macros. For example, assume that the characters PE are selected as a prefix for all data-names associated with a personnel file. The reserved word PERFORM will also be translated, causing a mistake of potentially major proportions. Selection of the Prefix macro name PE= avoids this problem, and also clarifies the intended translation of source words such as PE=EMPLOYEE-NUMBER AND PE=EMPLOYEE-NAME.

3.4 Nesting Word and Prefix Macros

Instead of repeating identical macro code in each of several macros, it is often convenient to have these macros substitute a special word (a word that cannot logically appear in the input stream) which acts as a call to a Word macro. The Word macro containing the common macro code is called a *macro subroutine*. Using a word substituted by one macro to invoke another macro is called *macro nesting.*

For example, a word substituted by a Prefix macro can match a short form defined by a Word macro. Give the following macros,

```
             WDP  BOH  :  BALANCE-ON-HAND
             PP   IN=& :  & OF INPUT-FILE
```

you can use IN=BOH in your source programs. The following occurs:

1. The Translator checks for Word macro matches; IN=BOH does not match a Word macro,.

2. The Translator searches for Prefix macro matches; IN=does match a Prefix macro. IN-BOH is expanded to:

```
                BOH OF INPUT-FILE
```

3. Since BOH matches a Word macro, it is expanded to BALANCE-ON-HAND, so that the output ultimately becomes:

```
                BALANCE-ON-HAND OF INPUT-FILE
```

As many as nine levels of macro nesting per source word are possible. Specifically, the rules for nesting are:

- Each qualifying word substituted from a Word macro model can invoke a single Prefix macro and a maximum of 8 additional levels of Word macros.

- Each qualifying word substituted from a Prefix macro model can invoke a maximum of 8 levels of Word macros.

Nesting String macros is described in the next section.

3.5 Summary - Writing Basic Macros

Listed below are the key points covered in this section.

- A macro consists of:

  type code
  division codes
  prototype
  separator (:)
  model

- Types are W (Word), P (Prefix), S (String), V (Verb), and U (Unverb).

- Division codes are I, E, D, and P. They indicate the COBOL divisions in which the macro is effective. The default is all divisions.

- The prototype consists of a word, a word prefix, or a string of words. The macro name is the first (or only) word of the prototype.

- The model provides substitution words, skeleton COBOL statements, CA-MetaCOBOL+ directives, or it can be omitted.

- The Word macro removes one word from the input source. It substitutes a word or string of words of source output, deletes the word, or performs other logic.

- The Prefix macro removes an input word with the same prefix as the prototype and any suffix. The suffix is saved as the symbolic operand &. The macro substitutes a sequence of output words that can include the suffix &.

- Macro names can be used in other macros as macro subroutine calls.

Exercise 2: Write Word and Prefix macros to simplify the underlined code listed below.

```
DATA DIVISION.
FILE SECTION.
FD OLD-MASTER-FILE  LABEL RECORDS ARE STANDARD
                    RECORDING MODE F
                    BLOCK CONTAINS 0 CHARACTERS
                    RECORD CONTAINS 80 CHARACTERS
                    DATA RECORD IS OLD-MASTER-RECORD.

01 OLD-MASTER-RECORD.
                    02 OLD-PLANT PIC 9(002).
                    02 OLD-DEPARTMENT  PIC 9(002).
                    02 OLD-ACCOUNT     PIC X(004).
                    02 OLD-AMOUNTPIC S9(05.  COMP-3.
                    02 OLD-DESCRIPTION PIC X(025).
                    02 OLD-SUFFIXPIX X(010).
                    02 FILLER    PIC X(052).
```

For exercise solutions, see Appendix A.

4. Getting Started - Writing String Macros

String macros are the most powerful macro type. They can identify extended strings of input source using prototypes that comprise constants and variables; they can be programmed to anticipate particular types of input and respond accordingly; and they can include macro models that define complex translation rules. Once you have learned to use String macros, you can begin to program more sophisticated CA-MetaCOBOL+ translations.

Format:

```
      S [division-code ...  ] prototype  :  model
```

S

Code the in the continuation column (cc. 7).

division-codes

Code the division-codes in columns 8-11 to specify that the macro applies to one or more COBOL divisions, and not the entire program.

4.1 String Macro Prototype

The String macro *prototype* begins with a macro name, just as a COBOL statement begins with a verb. It contains strings of intermixed words and symbolic operands.

As mentioned earlier, a symbolic operand represents a value acquired from input source. There can be as many as 15 unique symbolic operands specified in the String prototype. Each is represented by an ampersand (&) followed by a number from 1 to 15 (e.g., &1, &2, etc.). Each numbered symbolic operand can then be referenced in the macro model.

A String macro's symbolic operands represent words, such as COBOL reserved words, procedure names, data-names, alphanumeric literals (enclosed in quotation marks), and numeric literals.

For example, a String prototype to match a standard COBOL MOVE statement might appear as follows:

```
SP MOVE &1 to &15: ...
```

In order to understand the power of String macros, it is important to understand what CA-MetaCOBOL+ does when there is a match between source code words and a String macro prototype. Assume that a source program is loaded along with the prototype shown above. The source program includes the following:

```
$PD
...
  MOVE `TOTAL' TO TOTAL-LINE.
```

CA-MetaCOBOL+ searches the source program's Procedure Division as follows:

1. The source word MOVE is matched against the stored list of macro names. CA-MetaCOBOL+ finds that MOVE is defined as the name of the String prototype shown in the example. IF the MOVE were not defined as a macro name, it would be passed to the output, unaltered, as standard COBOL.

2. The next source word, the literal `TOTAL', is stored as the current value of &1, the first symbolic operand of the prototype.

3. The keyword TO of the prototype is then compared to the next source word. If unequal, CA-MetaCOBOL+ looks for <u>another</u> MOVE macro.

4. Since the comparison is equal, the next word, TOTAL-LINE, becomes the current value of &15.

5. The entire MOVE `TOTAL' TO TOTAL-LINE statement, therefore, is removed from the source. The action specified in the model (not yet shown) associated with MOVE &1 TO &15 is executed.

Anticipating Symbolic Operand Matches

Consider the following source statements in relation to the string macro prototype previously defined:

```
$PD
  MOVE ABLE OF BAKER (SS1) TO CHARLIE...
```

Although the above is a valid MOVE statement, it does not match the current MOVE prototype because the third source word, OF, does not match the third prototype word, TO.

To enable a match, possible qualifiers, subscripts, and literals must be anticipated in a String prototype. The table on the next page shows the codes that can be appended to symbolic operands to specify what type of source statements they can match.

Symbolic Operand	**Matches**

&n	The next word (a data-name or procedure-name without qualifiers, subscripts, or indexes: or a literal).
	
	Example: `ADD`
	`` `TOTAL' ``
	`+50`

&n(Q)	A data-name or procedure-name that may be qualified by the connectors OF or IN but may NOT be subscripted.
	
	Example: `COMPUTE-NEW-BALANCE`
	`COMPUTE-NEW-BALANCE OF SECTION-1`
	`COMPUTE-NEW-BALANCE IN SECTION-1`

&n(S)	A data-name that may be qualified, subscripted/indexed, or both.
	
	Example: `AMOUNT`
	`AMOUNT OF MASTER-RECORD`
	`AMOUNT OF MASTER-RECORD (SUB1)`

&n(L)	A literal or figurative constant ONLY.
	
	Example: `` `TOTAL' ``
	`+50`
	`SPACES`

&n(R)	A data-name that may be qualified, subscripted, indexed, and reference modified.
	
	Example: `NAME`
	`NAME OF ADDR-RECORD (SUB1)`
	`NAME OF ADDR-RECORD (SUB1)`
	`(4:20)`

&n(S,L)	A data-name that may be qualified and/or subscripted, or a literal.
	
	Example: `AMOUNT`
	`AMOUNT OF MASTER-FILE(SUB1)`
	`` `TOTAL' ``

&n(R,L)	A data-name that may be qualified, subscripted, and reference modified, or a literal.
	
	Example: `NAME`
	`NAME OF ADDR-RECORD`
	`(SUB1)(10:15)`
	`'UNDEFINED'`

The codes Q, R, and S cannot be specified if the anticipated source word represents a COBOL reserved word.

Taking the string prototype that was specified earlier, it can now be made more comprehensive by adding recognition codes. For example:

```
SP MOVE &1 TO &15:...
```

becomes

```
SP MOVE &1(S,L) TO &15(S):...
```

The keywords MOVE and TO are required. The first symbolic operand can be an identifier or a literal (including figurative constants). The second symbolic operand can only be an identifier.

4.2 String Macro Model

String macro models, like all other models, specify the action to take when source words match the prototype.

In its simplest form, the model of a String macro looks like a Prefix macro model. The String model, however, can substitute more than one symbolic operand; therefore, &1, &2, etc., are used to identify different symbolic operands.

String macro models, as well as Word and Prefix macro models, can do much more than has been shown so far. Data elements such as concatenations and variables can be employed. Directive expressions can be used to control macro processing. Once you have mastered the basic macros shown thus far, the next sections show how to extend the processing capabilities of macro models.

4.3 Uses of String Macros

The basic uses of String Macros include:

- New COBOL verbs can be defined that represent complex COBOL functions, such as internal table sorting, counter resetting, and free-format parameter analysis.
- COBOL sub-languages can be built for report writing, input validation, file match/merge, and other specialized functions from interrelated CA-MetaCOBOL+ verbs.
- Convenient interfaces can be written to data base management systems (such as CA-DATACOM/DB or IMS) and teleprocessing monitors (such as CICS).

- Utilities can be built for COBOL-to-COBOL conversion, standards auditing, source optimization, and many other functions, using interrelated String macros which examine and modify existing COBOL source programs.

Multiple-Variable Short Form Example

The following example provides a short form method for specifying the INSPECT...REPLACING LEADING...statement of COBOL. When the input source matches the macro ERL:

- The identifier INPUT-RECORD is stored as &7.
- The literal `0' is stored as &3.
- The literal `*' is stored as &5.

Note that the number identifying each symbolic operand must be unique, but not necessarily in ascending or consecutive sequence.

Note also that the current value for &7, INPUT-RECORD, is neither qualified nor subscripted, although the symbolic operand has been defined so that qualifiers and subscripts, if present, would be included as &7.

CA-MetaCOBOL+ Input:

```
SP      ERL &7(S) &3(L) &5(L)    :
        INSPECT &7 REPLACING LEADING &3 BY &5
 .  .  .
 PROCEDURE DIVISION.
 .  .  .
        ERL INPUT-RECORD `0' `*'.
 .  .  .
```

CA-MetaCOBOL+ Output:

```
 .  .  .
 PROCEDURE DIVISION.
 .  .  .
        INSPECT INPUT-RECORD
        REPLACING LEADING `0' BY `*'.
 .  .  .
```

New Verb Example

New verbs can be defined by CA-MetaCOBOL+ to relieve the applications programmer of having to write, for example, these statements:

```
ADD literal TO LINE-COUNT
IF LINE-COUNT IS GREATER THAN +56
  PERFORM PAGE-HEAD-ROUTINE.
WRITE PRINT-RECORD FROM record-name
  AFTER ADVANCING literal LINES.
```

This basic print function is one of the many functions frequently repeated in COBOL environments. Instead of writing out these functions, which are, in fact, groups of interrelated logical statements, programmers using CA-MetaCOBOL+ can write:

```
PRINT record-name BY literal.
```

The following example demonstrates a String macro which defines such a PRINT verb.

Notice that the standard COBOL statements required to increment the line-count, check for page-overflow, transfer to a new page routine, and finally, print with forms-control have been generated from a simple functional verb, PRINT. Only the logical values for the symbolic operands representing the record-name (D-LINE) and the literal (3) need to be specified.

CA-MetaCOBOL+ Input:

```
SP      PRINT &1(S) BY &2(L) :
        ADD &2 TO LINE-COUNT.
        IF LINE-COUNT IS GREATER THAN +56
          PERFORM PAGE-HEAD-ROUTINE.
        WRITE PRINT-RECORD FROM &1
          AFTER ADVANCING &2 LINES.


 .  .  .
 PROCEDURE DIVISION.
 .  .  .
        PRINT D-LINE BY 3.
```

CA-MetaCOBOL+ Output:

```
 .  .  .
 PROCEDURE DIVISION.
 .  .  .
        ADD 3 TO LINE-COUNT.
        IF LINE-COUNT IS GREATER THAN +56
          PERFORM PAGE-HEAD-ROUTINE.
        WRITE PRINT-RECORD FROM D-LINE
          AFTER ADVANCING 3 LINES.
```

4.4 Nesting String Macros

The previous section showed how the output from Word or Prefix macros can be used, in turn, by other Word or Prefix macros. String macros can also be nested with other macro types, with these limitations:

- Substituted words from a String macro can be used in a single Prefix macro and up to nine levels of Word macros.
- Substituted words from Word and Prefix macros may be used in a String macro BUT NOT as the macro name; the String macro name will only match a word from source input.
- Only one String macro can appear in the nested structure.

These rules mean that, for example, macro subroutine calls consisting of Word and Prefix macro names can be expanded while a more complex String macro is expanded, with little concern for exceeding built-in nesting limits.

See Appendix B for a macro nesting illustration.

4.5 Macro Interrelationships

Up to now, the primary emphasis has been on the relationship between source words and macros. A different perspective on macro writing can be gained by examining the relationships that can exist between macros.

In most situations, a programmer can use, for instance, both shorthand notation and more complex macros at the same time, and control the generation of COBOL code without concern for the conflicts between the macros.

There are, however, other situations. For example, consider the following macros, which are mutually exclusive when considering the source word PRINT:

```
WP   PRINT  :  PERFORM WRITE-A-LINE
PP   PR  :  & OF PAYROLL-FILE
```

What will happen to a source work PRINT--will it become INT OF PAYROLL-FILE or PERFORM WRITE-A-LINE?

This subsection describes:

- The priorities observed when attempting to match a source word to a macro prototype and thereby invoke execution of the macro model.

- The priorities of macro invocation when several macro prototypes qualify.
- The priorities of macro invocation of String macros with the same macro name, but otherwise different prototypes.

Macro Search Priorities

In case of a conflict among macro prototypes, the following rules apply:

1. Word and String macros take precedence over Prefix macros.

   Since Word and String macro prototypes are considered to be more specific than Prefix macros, CA-MetaCOBOL+ first matches each source word passed to macro translation that qualifies as a macro name against the stored Word and String macro names.

   If the source word matches a Word macro name, the corresponding macro model is invoked. If the source word matches a String macro name, CA-MetaCOBOL+ attempts to match the entire prototype to successive source words, and if a match occurs, the corresponding macro model is invoked.

   Only if the source word fails to match a Word or String macro prototype, does CA-MetaCOBOL+ attempt to determine if there is a match to a Prefix macro.

2. The macros loaded last take precedence.

   In all cases, the macro name search proceeds in the reverse order from which the macros were loaded. This means that the last macros loaded are the first to be examined for a match.

This search sequence means that a source word that explicitly invokes a Word or String macro:

- Will not look for a matching Prefix macro.
- Will not look further for additional Word or String macro matches.

Macro-name Override

The search sequence principle implies that if several macros have identical macro names, only the last one loaded is invoked. Therefore, to override or conceal a macro when the macro prototype consists only of a single word (the macro name), place a macro with the same name later in the input stream.

For example, consider the following identically named macros loaded into the CA-MetaCOBOL+ internal tables in the order shown:

```
WP PRINT : MODEL1
.   .   .
WP PRINT : MODEL2
.   .   .
```

The source word PRINT first looks for a matching Word macro or String macro name, the search proceeding from the last macro loaded. When the last PRINT macro loaded is found, a match occurs and the PRINT model (model2) is invoked. All but the last PRINT macro loaded are overriden and cannot be invoked.

In this example, either or both macros could have been defined as String-type, with identical results. Since both Word and String macros are matched at the same time, the last PRINT macro is invoked, regardless of type.

Macro Prototype Overrides

A prototype override can occur when multi-word String macros have the same macro name, but otherwise different prototypes. Since the macro prototypes are searched in reverse order, a macro is never invoked if a later prototype matches the same source words.

Consider the following macro definitions and Procedure Division source statements:

```
0100          SP PRINT &1(L):...
                .  .  .
0200          SP PRINT &1(S):...
                .  .  .
0500           PROCEDURE DIVISION.
                .  .  .
0700             PRINT IDENTIFIER (X)...
                .  .  .
0900             PRINT `LITERAL'...
```

In this example:

1. The PRINT statement on line 700 attempts to match the PRINT macro defined on line 200. Since IDENTIFIER(X) qualifies as a match to the symbolic operand notation &1(S), the associated macro model is invoked.

2. In like manner, the PRINT statement on line 900 attempts to match the PRINT macro defined on line 200. The match fails because `LITERAL' is a non-numeric literal and does not qualify as a match to &1(S).

3. The search continues with an attempt to match line 900 with the macro defined on line 100. `LITERAL' qualifies as a match to &1(L), and the macro on line 100 is invoked.

4. An override occurs, however, if the macro defined on line 200 is defined as follows:

   ```
   0200    SP PRINT &1(S,L) : ...
   ```

   Here, both print statements match the symbolic operand &1(S,L), so the macro defined on line 200 is always invoked, and the macro defined on line 100 is overridden.

Since the macro search occurs in the reverse sequence from which the macros are loaded, the above situation can be controlled by adhering to the following rule:

Load the macro of the most selective prototype last.

4.6 Summary - Writing String Macros

Listed below are the key points covered in this section.

- String macros are used to make complex substitutions and translations; they are the most powerful macro type.

- The String macro prototype can specify a string of intermixed words and up to 15 symbolic operands (&1-&15).

- Symbolic operands in a String macro prototype can include codes to anticipate the kinds of input they will match. Codes are:

  Q matches only a data-name or a procedure-name that may be qualified.
  R matches only a data-name that may be qualified, subscripted, and reference modified.
  S matches only a data-name or a procedure-name that may be qualified and/or subscripted.
  L matches only an alphanumeric literal, numeric literal, or a figurative constant.
  R,L matches either a qualified, subscripted, and reference-modified data-name, or a literal.
  S,L matches either a qualified and/or subscripted data-name or a literal.

- The model of a String macro can include more than one symbolic operand, and a symbolic operand can be used more than once.

- Word macros can be nested up to nine levels deep. One Prefix macro can take up one of these levels. A String macro can "call" the nested structure.

- In cases of conflict between prototypes, Word and String macros take precedence over Prefix macros. In cases of further conflict, the macros loaded last take precedence; therefore, place the macro with the most selective prototype last.

Exercise 3: The COBOL SEARCH verb does not initialize the index into the table to be searched. Write a String macro that finds each occurrence of:

```
SEARCH EVERY table-item index-name
```

where table-item is defined with an OCCURS clause, and index-name is the name of the first index defined by the item. The macro model initializes the index to 1 and outputs a SEARCH table-item statement:

```
SET index-name TO 1
SEARCH table-item
```

As a second exercise, rewrite the FD entries in Exercise 2 using a String macro. (You can nest Word macros within the String macro.)

For exercise solutions, see Appendix A.

5. Getting Started - Model Programming

So far, this guide has discussed how to remove input source words using macro prototypes and replace them in output source using macro models. While such capabilities are essential, they do not reflect the *programmable* nature of CA-MetaCOBOL+'s macro language.

With programmable macro language, the user can control the analysis of source code and the generation of output COBOL statements.

The last section suggested that the macro model can be used for more than simple word substitution. The model can also define the actions to take when the model is invoked.

Three terms are used frequently throughout the rest of this manual to describe how to control macro execution and the resulting output:

- A *substitution word*, a term already introduced, refers to any word in a macro model that is either output directly or is replaced by a matching prototype.
- A *directive* is a special word, embedded only within a macro model, which causes the Translator to take explicit action, but which does not appear in the output.
- A *directive expression* is a directive followed by one or more operands.

The following example combines a substitution with a directive:

```
WD  -WS : &A WORKING-STORAGE SECTION.
```

The header WORKING-STORAGE SECTION. is the substitution text. &A is a directive that explicitly forces the next word into Area A of the generated output. Note that the &A directive does not appear in the output: it causes the substituted word, WORKING-STORAGE, to appear in Area A.

As an introduction to how to program macro models, this section describes basic functions that can be specified in macro models and some rules for macro programming. These include:

- Building CA-MetaCOBOL+ words.
- Displaying diagnostic messages.
- Forcing Area A or Area B alignment.
- Placing terminating periods.

5.1 Building Words

CA-MetaCOBOL+ defines a word in the same manner as COBOL. However, one word form is unique to CA-MetaCOBOL+. Within macro models, you can build a single word from one or more words, literals, or the current values of symbolic operands or variables. (Variables are discussed in the next section.)

This process of word building is called *concatenation*-i.e., linking together. Concatenation within macro models allows you to build unique data-names, procedure-names, literals, and diagnostic messages and to fill many other translation requirements.

Format:

```
&(expression &)
&(Q expression &)
```

&(, &), and **&(Q**

The &(, &), and &(Q are CA-MetaCOBOL+ directives. They instruct the Translator to concatenate the elements in the enclosed *expression*. &(or &(Q begins the concatenation; &) ends it.

The expression can consist of constants (literals, punctuation, names, etc.), symbolic operands, and variables, separated by one or more blanks. These elements are combined by the Translator with all embedded blanks removed. The result is a single word.

Use the first format above if the result of the concatenation IS NOT to be bounded by quotes. Use the second format above if the result IS to be bounded by quotes (as an alphanumeric literal). Please note:

- The directive &(and &(Q must be preceded and followed by one or more blanks.
- The directive &) must be preceded by one or more blanks and must be followed by one or more blanks or a terminating punctuation character.

As an example of concatenation, the logical result of the expression,

```
           &( A B C &)
```

becomes,

```
           ABC
```

because the concatenation beginning with &(does NOT build a word bounded by quotes.

Conversely, the result of the expression,

```
           &(Q A B C &)
```

becomes,

```
           `ABC'
```

because the concatenation beginning with &(Q builds an alphanumeric literal. Please note:

- Symbolic words are translated to their current values. If the symbolic word contains a multi-word current value, only the first word of the current value is included.
- Alphanumeric literals are considered to be words. Bounding quotes are removed, but the entire contents are retained. Spaces contained within quotes are NOT removed.

Word-Building Example

As an example of a word-building application, concatenation can be used with Prefix macros to expand short prefixes into more meaningful terms. This is accomplished by concatenating the current value of the symbolic operand (&) and the descriptive term. (For example, input source M-RECORD yields output source MASTER-RECORD.)

CA-MetaCOBOL+ Input

```
PDP M-  :  &( MASTER- & &)
 .  .  .
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 M-RECORD.
     02 M-BALANCE ...
 .  .  .
 PROCEDURE DIVISION.
 .  .  .
     MOVE ZERO TO M-BALANCE ...
```

CA-MetaCOBOL+ Output:

```
.   .   .
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MASTER-RECORD.
     02 MASTER-BALANCE ...
.   .   .
PROCEDURE DIVISION.
.   .   .
     MOVE ZERO TO MASTER-BALANCE ...
```

When building a substitution word, a terminating period is a *separate* word. It should be placed to the right of the &) so that CA-MetaCOBOL+ does not substitute multiple, consecutive periods.

5.2 Generating Diagnostics

There are many conditions which you may wish to diagnose during translation. Diagnostics are commonly used for the following purposes:

- To flag COBOL verb usage which is in violation of internal standards.
- To flag explicit references to critical data-names and procedure-names during program maintenance.
- To indicate invalid macro usage in the input source.
- To advise of source modification, statement inefficiency, or standards violation during COBOL-to-COBOL conversion, source optimization, or standards auditing functions.
- To trace logical flow and current values of symbolic words during complex macro debugging.

Based on the CA-MetaCOBOL+ options set prior to macro processing, diagnostic messages can be displayed on the source input listing, on the COBOL output listing, or on both.

&NOTE is a CA-MetaCOBOL+ directive used in a macro model to display a macro-defined message.

Format:

```
&NOTE diagnostic-word
```

&NOTE
displays a diagnostic-word when executed in a macro model.

diagnostic-word
The diagnostic-word can be a constant, symbolic operand, concatenation, or variable. (Variables are explained in the next section.)

The diagnostic-word becomes the diagnostic message but is NOT substituted as an output word. Please note:

- The maximum length of the diagnostic message is 64 characters; any excess is truncated from the right.
- If a symbolic operand is used as the diagnostic-word, only the first word of the current value is displayed.

Diagnostic Example

Assume that two messages indicating a standards violation are to be displayed on the CA-MetaCOBOL+ Input Listing following each ALTER verb found in the Procedure Division. The first message contains the ALTERed procedure-name, and the second contains an explanatory message, as follows:

CA-MetaCOBOL+ Input Listing:

```
          SP   ALTER &1(Q)  :
               ALTER &1
               &NOTE `ALTERED - STANDARDS ERROR'
           .  .  .
           PROCEDURE DIVISION.
           .  .  .
               ALTER PARA-A TO ...
****NOTE N99 PARA-A
****NOTE N99 `ALTERED - STANDARDS ERROR'
               .  .  .
               ALTER PARA-C TO ...
****NOTE N99 PARA-C
****NOTE N99 `ALTERED - STANDARDS ERROR'
               .  .  .
```

Notice that two &NOTE directive expressions are required to display what is essentially one message.

Had the &NOTE been INCORRECTLY specified as,

```
          &NOTE &1 `ALTERED-STANDARDS ERROR'
```

only the current value of &1 would have been considered part of the diagnostic and displayed on the input listing. The literal, being outside the directive expression, would have been substituted in the output source, causing a serious logical error.

5.3 Controlling Area A and Area B Alignment

In general, the CA-MetaCOBOL+ Translator aligns output words in the same area as it finds them in the input source. This means that words that appear in Area A of the input source appear in Area A of the output source. Similarly, words that appear in Area B of the input source appear in Area B of the output source.

CA-MetaCOBOL+ makes the following assumptions in aligning output:

- If the input source word is not replaced, it will appear in the same area in the output as in the input.
- If the input source word matches the macro prototype, placement depends on the following:
- A substituted word that begins in Area B of the model is aligned in the same area in the output source as in the input source.
- A substituted word that begins in Area A of the model is aligned in Area A of the output source no matter where it appears in the input source.

Area A Alignment Example

In this example, although the abbreviation CSEC is found in Area B of the input source, the three Area A words, CONFIGURATION, SOURCE-COMPUTER, and OBJECT-COMPUTER, are forced into Area A because they are defined in Area A of the macro model.

CA-MetaCOBOL+ Input:

```
WE    CSEC   :
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.   IBM-370.
 OBJECT-COMPUTER.   IBM-370.
 .  .  .
 $ID
 .  .  .
 $ED CSEC
 .  .  .
```

CA-MetaCOBOL+ Output:

```
          IDENTIFICATION DIVISION.
          .  .  .
          ENVIRONMENT DIVISION.
          CONFIGURATION SECTION.
          SOURCE-COMPUTER.   IBM 370.
          OBJECT-COMPUTER.   IBM-370.
          .  .  .
```

You can guarantee Area A alignment of substituted words by defining substitution words in Area A of Word, Prefix, and String macro models. However, the preferred method for controlling Area A and Area B alignment is provided by two format control directives that can be used in macro models.

Area A Directive Format:

```
     &A
```

&A

The &A directive, if executed in a macro model, causes the next word to begin a new line of output in Area A.

&A Alignment Example

The following demonstrates two methods for forcing Data Division section headers substituted by Word macros into Area A:

CA-MetaCOBOL+ Input:

```
          WD    -FS   :
           FILE SECTION.
          WD    -WS   :  &A WORKING-STORAGE SECTION.
          .  .  .
           $DD  -FS
           .  .  .
           -WS
           .  .  .
```

CA-MetaCOBOL+ Output:

```
          DATA DIVISION.
          FILE SECTION.
          .  .  .
          WORKING-STORAGE SECTION.
          .  .  .
```

Area B Directive Format:

```
&B
```

&B

The &B directive, if executed in a macro model, causes the next word to begin a new line of output in Area B. If it precedes a substitution word, &B cancels any pending Area A indicators.

Area B Alignment Example

The following String macro uses &B to format clauses in multiplication statements.

CA-MetaCOBOL+ Input:

```
SP    ROUND &1 OF &2 TIMES &3 ERROR &4    :
      &B MULTIPLY &2 BY &3
           &B GIVING &1 ROUNDED
           &B ON SIZE ERROR GO TO &4

  .  .  .
 $PD
      ROUND PRODUCT OF MULTIPLIER TIMES MULTIPLICAND
      ERROR ERROR-ROUTINE.
```

CA-MetaCOBOL+ Output:

```
.  .  .
PROCEDURE DIVISION.
     MULTIPLY MULTIPLIER BY MULTIPLICAND
       GIVING PRODUCT ROUNDED
       ON SIZE ERROR
       GO TO ERROR-ROUTINE.
```

Exercise 4: Define two macros to generate the following:

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.

WORKING-STORAGE SECTION.
```

For exercise solutions, see Appendix A.

5.4 Where to Place Terminating Periods

Terminating periods can be translated into the COBOL output program from either the input source or the macro model. If consecutive periods occur, only the first period is generated.

In the example of the CSEC macro shown previously, periods are placed in the macro model where required by COBOL, including at the end of the OBJECT-COMPUTER paragraph. The last word placed in the output as a result of the translation is the period. The programmer could, therefore, specify "CSEC" or "CSEC.".

One of the most common errors in macro writing is the incorrect use of periods within macro models. In order to avoid such errors, a good rule of thumb is to specify Area A alignment and periods in macro models only where required or where explicit macro usage dictates (as in the CSEC macro). In all other cases, allow the source programmer to control Area A alignment and period placement at the source level.

Terminating Period Example

Consider placement of a period in the model of the following macro, defined to replace PAL.

CA-MetaCOBOL+ Input:

```
WP   PAL:   PERFORM PRINT-A-LINE
 .   .   .
 $PD
 .   .   .
    IF ERROR-SWITCH = `1'
    PAL
    GO TO READ-A-CARD.
 .   .   .
```

CA-MetaCOBOL+ Output:

```
.   .   .
PROCEDURE DIVISION.
.   .   .
    IF ERROR-SWITCH = `1'
    PERFORM PRINT-A-LINE
    GO TO READ-A-CARD.
.   .   .
```

Notice that if the PAL macro had been written incorrectly as,

```
  WP  PAL : PERFORM PRINT-A-LINE.
```

it would have introduced a logical error. The PAL would have been translated with a terminating period, and the GO TO statement would no longer be part of the conditional statement.

5.5 Model Programming - The CA-MetaCOBOL+ Macro Language

The ability to program a macro model means that CA-MetaCOBOL+ offers true macro language capabilities. As will be discussed in the following sections, these capabilities offer a wide variety of programming functions. They include:

- Using Variables - How to use alphanumeric and numeric variables for the storage of words or symbolic operands, as work areas for internal computation, or as switches. These variables can be modified and interrogated within a macro model and made available to other macros. In this manner, information can be passed from macro to macro.
- Defining Branches and Control Structures - How to define logical destination and to transfer control to these destinations either conditionally or unconditionally. Also, how to use structured selection and repetition control logic in macro model programming.
- Acquiring Input for the Model - How to acquire attributes of input source data, and how to retrieve data items from the COBOL Data Division. Model processing decisions can be based on exactly what kind of input source the macro model is reading.
- Parsing Input Storage - How to acquire additional source words from the input stream following a String macro call. This facility allows the macro model to control input processing when the input source may be too variable for a prototype to anticipate.
- Controlling Output - How to generate unformatted text, such as comments, compiler directives, or JCL, how to control the alignment of output source, and how to generate output statements to a location other than that of a macro call.

5.6 Summary - Model Programming

Listed below are the key points covered in this section.

- A CA-MetaCOBOL+ macro model defines the actions to take or the words to generate when the macro is invoked. It can include:

  Substitution words that appear in the output.

  Directives that affect output but do not appear in the output.

  Directive expressions that include a directive and one or more operands.

- Concatenation builds a CA-MetaCOBOL+ word in a macro model from constants, symbolic operands, and variables.

  The directives &(and &) delimit a concatenation without surrounding quotes.

  The directives &(Q and &) delimit a concatenation with surrounding quotes.

- &NOTE can be used in macro models to output a diagnostic word.

- Substituted words are generally aligned in the same area in source output as in input; placing the first word of the macro model in Area A forces substituted words to begin in Area A.

  To guarantee Area A alignment of source output, use the &A directive in the macro model.

- &B forces a new line of output in Area B.

- Specify COBOL terminating periods in macro models only when required. In general, assume that the source programmer will supply them.

Exercise 5: Write a String macro to diagnose each ALTER verb found in the Procedure Division with a single diagnostic that contains the altered procedure-name, as:

```
   procedure-name ALTERED-STANDARDS ERROR
```

Remember, the &NOTE directive expression can include a concatenation construction.

For exercise solutions, see Appendix A.

6. Model Programming - Using Variables

The CA-MetaCOBOL+ programmer can make use of variables in macro models. The relationship of variables to a CA-MetaCOBOL+ macro model is similar to that of an elementary data item to a COBOL program. Variables are used to store data to be processed by the macro in which they are contained or by another macro.

Variables are similar to symbolic operands (such as & for Prefix macros and &1 - &15 for String macros, described earlier). Both variables and symbolic operands store the current values of changeable data. However, where symbolic operands generally represent words acquired from source input, variables are generally defined in macro models by the macro writer.

Unlike symbolic operands, variables are named and assigned picture types. They can be initialized, indexed and can be modified by a statement which serves the same function as a MOVE, an arithmetic statement, or a shift. Often a symbolic operand is stored in a variable for further processing.

This section describes:

- How to define variables.
- Initializing variables using &INIT.
- Referencing variables once they are defined.
- Using &SET and &SETR to modify variables.

6.1 Defining Variables

A variable must be defined before it can be used. A group of reserved variables are predefined by CA-MetaCOBOL+, but most variables are defined simply by naming them with an access type and picture type. CA-MetaCOBOL+ supports these variable types:

- Alphanumeric and numeric variables.
- Symbolic operand storage variables.
- Boolean (true/false) variables.
- Predefined system variables.

The directive expressions used to define these variable types are described next.

Alphanumeric and Numeric Variables

CA-MetaCOBOL+ alphanumeric and numeric variables can be used as word storage areas, as work areas for computation, and as indicators to control the logic of macro models. They are defined by the following directive expression:

Format:

```
{&EXTERN}
{&GLOBAL}  &Vname[(integer)] { = literal}  picture
{&LOCAL }                    {   NULL   }
```

&EXTERN, &GLOBAL, or &LOCAL
is the access type and defines how other macros can reference the variable.

Variable Type	**Can Be Referenced By**	**Must Be Unique Within**
External	All macros in all regions.	All regions.
Global	All macros in one region.	A single region.
Local	Only the macro in which it is defined.	A single macro.

See the *Macro Facility Reference* for a description of regions. Please note:

- If you define a local variable, you cannot reference an external or global variable with the same name in the macro model containing the local definition.
- If you define a global variable, you cannot reference an external variable with the same name in the macro region containing the global definition.

&Vname

is the name of the variable. &Vname can be up to 30 alphanumeric characters (including the leading &V) with no embedded spaces, punctuation, or quotation marks. (See the "Word" definition in the Glossary.)

(integer)

The optional integer may be used to define an indexed variable (variable table). Integer, in parentheses without intervening spaces, specifies the number of occurrences of the variable.

literal or NULL

Every occurrence of a variable can be initialized during macro loading by the value literal or NULL. Please note:

- The literal must be valid for the class of variable defined; that is, an alphanumeric variable value must be bounded by quotation marks and a numeric variable value must be an integer.
- Unless otherwise specified, the initial value of a numeric variable is a single 0, while the initial value of an alphanumeric variable is a single blank. The values repeat for all occurrences.
- NULL can be specified ONLY for alphanumeric variables and represents "no value" (the variable is empty). Other directives can set a variable to, or test for, the NULL condition. A variable with a NULL value which is output takes no space in the output.

Picture

defines the class and the maximum length of each occurrence of the variable. As in COBOL, picture can be specified as alphanumeric, in the form X[...X] or X(n), or unsigned numeric integers, in the form 9[...9] or 9(n). Please note:

- No more than 128 alphanumeric characters or 9 digits (including the minus sign if negative) can be defined and stored.
- Although defined as unsigned, numeric variables can assume a current value less than zero during translation.
- All current values, both alphanumeric and numeric, are stored left-aligned in variables so that COBOL output does not generate insignificant spaces or zeros. The picture, therefore, defines the *maximum capacity* of a variable. The actual length of the current value of a variable is the number of characters or digits stored.

Example

Two variables are defined here. Global variable &VNAME is defined as having a maximum of twelve alphanumeric characters with an initial value of NULL. Local variable table &VTABLE is defined as four 3-digit numeric fields, each with an initial, default value of a 0:

```
&GLOBAL  &VNAME  =  NULL X(12)

&LOCAL  &VTABLE(4)  999
```

Symbolic Operand Storage Variables

Because symbolic operands may need to be held before processing, CA-MetaCOBOL+ provides a symbolic operand storage variable. This variable is used to reserve an area large enough to hold all the words in a symbolic operand. Separate words are not concatenated. Refer to the *Macro Facility Reference* for detail.

Boolean (True/False) Variables

A special type of variable, the Boolean variable, can be used in macro models as on/off, yes/no switches. This variable type can be set to one of two values: true or false. It must be defined prior to any reference to its name.

Format:

```
{&EXTERN}
{&GLOBAL}  &Bname = {TRUE }
{&LOCAL }           {FALSE}
```

&B and **TRUE** or **FALSE**

Like &V variables, &B variables must be defined as External, Global, or Local. &Bname can have a maximum of 30 contiguous alphanumeric characters, including the leading &B, and may not contain embedded blanks, quotation marks, or punctuation. (See the "Word" definition in the Glossary.) The value is set by the equal-sign to either true or false.

Example

```
&LOCAL  &B-FIRST-TIME  =  TRUE

&GLOBAL  &B-EOF  =  FALSE
```

External Variables

There are several names for variables which are predefined as external by the CA-MetaCOBOL+ Translator.

Variable Name	Corresponding Option	Default Value
&VSOURCE	SOURCE=x	blank
&VDIALECT	DIALECT=x	as installed
&VUPSI1 - 8	UPSIn=x	blank

Within macros, these reserved variables can be tested, but not defined or modified. They are predefined as external, alphanumeric variables containing the values of translation-time options which can be used to control the translation process.

At installation or as a translate-time option, &VSOURCE may be set to indicate the compiler level of the input source program. &VDIALECT indicates the compiler level of the output source program. The &VUPSIn variables are user-defined to test specific conditions. (See the CA-MetaCOBOL+ *User Guide* for a description of translate-time options.)

6.2 Initializing Variable Tables

As explained earlier, a variable table, or indexed variable, can be automatically initialized to a default value or initialized by the programmer to an explicit value. This assigns the same value to all occurrences of the variable. You can also initialize different occurrences of the variable to different values. Use the &INIT and &IEND directives.

Format:

```
&INIT   {literal}
        {NULL   } .  .  .  &IEND
```

&INIT and **&IEND**
The &INIT and the &IEND directives delimit a list of values (literals or NULLs) to assign to the most recently defined variable table.

literal and **NULL**
are explained in Section 6.1, "Defining Variables."

Refer to the *Macro Facility Reference* for details.

6.3 Referencing Variables

Once you have defined a variable in a macro model, it can be referenced by name within other CA-MetaCOBOL+ directive expressions or as substitution words. For example,

```
&NOTE  &VNOTE
```

outputs the current value of &VNOTE.

Boolean variables are referenced by name in conditions and can be negated by NOT. For example:

```
&IF  &B-FLAG
&IF  NOT  &B-FLAG
```

If the variable has been defined as a table, reference must include the index. The index can be either a literal or a numeric variable containing a positive integer no greater than the number of occurrences of the variable. For example, the fourth occurrence of &VINDX is indicated as:

```
&VINDX(4)
```

The following refers to the occurrence of &VTABLE indicated by the current value of the numeric variable &VX:

```
&VTABLE(&VX)
```

Note: A variable cannot be referenced before it is defined.

6.4 Modifying Variables

The current values of alphanumeric, numeric, and Boolean variables can be modified by the &SET and &SETR directive expressions.

6.4.1 Data Move - &SET Format 1

```
      &SET &Vname = sending-item
```

```
Format 1 of the &SET directive expression can be compared to a
COBOL MOVE command.
```

&Vname

The receiving variable &Vname assumes the current value of the *sending-item* beginning at the left-most character of both fields.

sending-item

The sending-item can be one of the following:

Variable
Symbolic Operand
Concatenation
Literal
Symbolic-Operand Attribute (see Section 8)
NULL

Please note:

- If the current value of the sending item is longer than the maximum capacity of &Vname, excess characters or digits are truncated from the right.
- Only the first word of the sending item is moved, even if it is a multi-word symbolic operand.
- If a non-numeric literal is stored in an alphanumeric variable, the bounding quotes are stripped.
- If an alphanumeric variable is &SET to NULL, it contains no data. Only alphanumeric variables can be set to NULL.
- If &Vname is a numeric variable, the sending item must be an integer.

Example

The following variable definitions and &SET directives place `ABC' (`D' is truncated) in &VA and 3 (not 003) in the second occurrence of &VB:

```
          &LOCAL &VA X(3)
          &LOCAL &VB(5) 9(3)

          &SET &VA = `ABCD'
          &SET &VB(2) = 3
```

New Verb Example

The following macro was defined by the New Verb Example in Section 4. It translates the source statement,

```
           PRINT record-name BY literal
```

into the COBOL statements,

```
           ADD literal TO LINE-COUNT
           IF LINE-COUNT IS GREATER THAN +56
               PERFORM PAGE-HEAD-ROUTINE.
           WRITE PRINT-RECORD FROM record-name
               AFTER ADVANCING literal LINES.
```

The choice of PAGE-HEAD-ROUTINE, +56, and PRINT-RECORD was arbitrarily made to simplify the problem.

Assume that the original prototype is still valid. The page-heading routine, depth-of-page, and printer-record can be specified as variables which can be passed from a statement called at the logical place in the Procedure Division where one would normally OPEN the printer file and print the first page heading. The macro called generates the proper OPEN and PERFORM, then saves additional parameters for reference by subsequent PRINT macro calls.

Two macros are required.

1. The OPEN-PRINT macro substitutes logical values for symbolic operands &1 (LIST-FILE) and &4 (NEW-PAGE-HEAD) directly.

   OPEN-PRINT stores LIST-AREA, 60, and NEW-PAGE-HEAD in variables for subsequent substitution by the PRINT macro.

2. The PRINT macro substitutes five items: &1 (D-LINE) and &2 (3) from the PRINT prototype itself and three items from the &GLOBAL variables defined in the OPEN-PRINT macro.

CA-MetaCOBOL+ Input:

```
SP     OPEN-PRINT &1 USING &2 DEPTH &3(L) OVERFLOW &4  :
       OPEN OUTPUT &1.
       PERFORM &4.
       &GLOBAL &VRECORD X(30)                 /* RECORD-NAME
       &GLOBAL &VDEPTH 9(2)                   /* DEPTH OF PAGE
       &GLOBAL &VOFLO X(30)                   /* PAGE HEADING
       &SET &VRECORD = &2
       &SET &VDEPTH = &3
       &SET &VOFLO = &4

SP     PRINT &1(S) BY &2(L)  :
       ADD &2 TO LINE-COUNT.
       IF LINE-COUNT IS GREATER THAN &VDEPTH
           PERFORM &VOFLO.
       WRITE &VRECORD FROM &1
           AFTER ADVANCING &2 LINES.

.  .  .
PROCEDURE DIVISION.
       OPEN-PRINT LIST-FILE USING LIST-AREA
         DEPTH 60 OVERFLOW NEW-PAGE-HEAD.
       .  .  .
       PRINT D-LINE BY 3.
```

CA-MetaCOBOL+ Output:

```
.  .  .
PROCEDURE DIVISION.
       OPEN OUTPUT LIST-FILE.
       PERFORM NEW-PAGE-HEAD.
       .  .  .
       ADD 3 TO LINE-COUNT.
       IF LINE-COUNT IS GREATER THAN 60
          PERFORM NEW-PAGE-HEAD.
       WRITE LIST-AREA FROM D-LINE
          AFTER ADVANCING 3 LINES.
```

6.4.2 Computed Value - &SET Format 2

```
                        {+}
&SET &Vname = item1  {-} item2
                        {*}
                        {/}
```

Format 2 of the &SET directive expression is the computational format.

&Vname
is the result and must be defined as numeric.

item1
is the augend (+), minuend (-), multiplicand (*), or dividend (/), and must be numeric.

item2
is the addend (+), subtrahend (-), multiplier (*), or divisor (/), and must be numeric.

Either item can be any of the following:

Variable
Symbolic Operand
Literal
Symbolic Operand Attribute

Please note:

- &Vname, although unsigned, can contain a value less than zero. If an arithmetic overflow occurs, truncation of the numeric result is from the *right*. Thus, care should be taken to ensure that the receiving fields are large enough to hold the largest computation possible. The following example illustrates the problem that can occur:

```
&LOCAL &VN 9
&SET &VN = 9
&SET &VN = &VN + 1
```

The result (10) is truncated from the right so that it fits the one-digit field defined for &VN, leaving the value of &VN equal to 1.

- Fractional results are truncated.

- Only one arithmetic operator is permitted with each &SET computation.

Example

The following variable definitions and &SET directives increment the current value of &VA by 1, decrement the current value of &VS by 1, compute the value of &VM as the product of &VA multiplied by &VS, and compute &VD as the quotient of &1 divided by 2:

```
&LOCAL &VA 9(3)               /* ADD
&LOCAL &VS 9(3)               /* SUBTRACT
&LOCAL &VM 9(3)               /* MULTIPLY
&LOCAL &VD 9(3)               /* DIVIDE

&SET &VA = &VA + 1
&SET &VS = &VS - 1
&SET &VM = &VA * &VS
&SET &VD = &1 / 2
```

6.4.3 Character Shift - &SET Format 3

```
&SET  &Vname  =  sending-item  %  index
```

&Vname

Format 3 of the &SET directive expression sets alphanumeric variable *&Vname* to selected characters from *sending-item.* The sending-item can be an alphanumeric or numeric variable or a symbolic operand (&1 - &15).

%

is required.

index

is a numeric index, or pointer, which refers to the "nth" character of the sending-item as the first character to be transferred to &Vname. The index can be a variable, symbolic operand, or literal containing a value of 1 through 128.

Please note:

- If the index equals one, the logical result in &Vname is equivalent to a Format 1 &SET.
- If the sending-item contains more than one word, only the first word can be accessed by this format of the &SET directive expression.
- If the index is greater than the number of characters in the sending-item, the logical result in &Vname is equivalent to a Format 1 &SET to NULL.

Example

Assuming a current value of "ABLE OF BAKER" in &1, the following variable definitions and &SET directives place "ABLE" in &VA, "BLE" in &VB, and &VC becomes NULL (contains no value):

```
&LOCAL &VA X(4)
&LOCAL &VB X(4)
&LOCAL &VC X(4)

&SET &VA = &1 % 1
&SET &VB = &1 % 2
&SET &VC = &1 % 5
```

6.4.4 Word Shift - &SET Format 4

```
&SET  &Vname  =  &n  #  index
```

&Vname

Format 4 of the &SET directive expression sets alphanumeric variable *&Vname* to a selected word from *&n*, a symbolic operand (&1 - &15).

#

is required.

index

is a numeric index, or pointer, which refers to the "nth" word of &n as the word to be transferred to &Vname. The index can be a variable, symbolic operand, or literal. Please note:

- If the index equals 1, the logical result in &Vname is equivalent to a Format 1 &SET.
- If the index is greater than the number of words in &n, the logical result stored in &Vname is equivalent to a Format 1 &SET to NULL.

Example

Assuming a current value of "ABLE OF BAKER" in &1, the following variable definition and &SET directives place "ABLE" in the first occurrence of &VA, "OF" in the second, "BAKER" in the third, and the fourth occurrence becomes NULL.

```
&LOCAL &VA(4) X(30)
&SET &VA(1) = &1 # 1
&SET &VA(2) = &1 # 2
&SET &VA(3) = &1 # 3
&SET &VA(4) = &1 # 4
```

6.4.5 Boolean Switch Set - &SET Format 5

The &SET directive alters the truth value of a Boolean variable.

```
&SET &Bname = {TRUE }
              {FALSE}
```

&Bname
is the name of a Boolean variable.

TRUE and FALSE
are the only values available.

Example

This example uses an &IF-&ENDIF construct (described in Chapter 7) to test the status of Boolean variable &B-FIRST-TIME. If it is true, the &SET alters the value to false. Otherwise, the variable is not altered.

```
&LOCAL &B-FIRST-TIME = TRUE
     .  .  .
&IF &B-FIRST-TIME
     &SET &B-FIRST-TIME = FALSE
     .  .  .
&ENDIF
```

6.4.6 Register Set - &SETR Directive Expression

The &SETR directive expression provides access to certain translate-time conditions. It can be used to store special registers in variables for subsequent analysis.

Format:

```
&SETR  &Vname  =  register-name
```

&Vname
is the receiving variable.

Register-name
identifies the special register.

While all of the special registers are described in the *Macro Facility Reference*, a few are discussed below to illustrate the type of information available via &SETR.

Register-name	**Picture**	**Meaning**
DATE	X(8)	Current date.
ID	X(8)	Contents of columns 73-80 of current input record.
LINE	9(6)	Input line number.
SEQ	X(6)	Input line sequence number (col. 1-6).
TIME	X(8)	Time of day (hh:mm:ss).
VAR	X(30)	Single word of external information from translate-time VAR= option.

The values of DATE, TIME, and VAR are constant throughout the translation. The values of ID, LINE, and SEQ reflect the most recent input record. (Section 9.4 describes the use of &SETR in analyzing control words acquired by &GET.)

Date-Written Example

As an example of &SETR directive use, a simple Word macro can be written to build the DATE-WRITTEN paragraph, as follows:

CA-MetaCOBOL+ Input:

```
WI      DW   :
 DATE-WRITTEN.
       &LOCAL &VDW X(8)
       &SETR &VDW = DATE
       &VDW.
 .  .  .
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  ...
 .  .  .
 DW
 .  .  .
```

CA-MetaCOBOL+ Output:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   ...
.  .  .
DATE-WRITTEN.  01/11/92.
.  .
```

6.5 Summary - Using Variables

Listed below are the key points covered in this section.

- CA-MetaCOBOL+ variables allow you to store information for use in macro models.
- &Vname variables must be defined as either alphanumeric or numeric with a maximum size and as External, Global, or Local. They can be indexed and can be initialized to any values compatible with their type.
- &Bname Boolean variables must be defined as External, Global, or Local. They can only have the value true or false.
- Symbolic operand storage variables are used to hold the current values of symbolic operands temporarily.
- CA-MetaCOBOL+ also provides predefined system variables.
- Variables are referenced by name to access their current value. &Vname and &Bname variables must be defined before being referenced.
- &SET directives are used to modify variables.
- &SETR accesses translate-time registers.

Exercise 6: Define a CA-MetaCOBOL+ String macro that will store file names taken from Environment Division SELECTs in a table. The table must be available for subsequent reference by other macros in the region. It must be large enough to hold up to nine file names.

CA-MetaCOBOL+ input might appear as:

```
  .  .  .
  ENVIRONMENT DIVISION.
  FILE-CONTROL.
  SELECT INPUT-FILE ASSIGN TO.  .  .
  SELECT MASTER-FILE ASSIGN TO.  .  .
  SELECT REPORT-FILE ASSIGN TO.  .  .
  .  .  .
```

Your macro should build a table containing the following entries:

```
  INPUT-FILE
  MASTER-FILE
  REPORT-FILE
.  .  .
```

Note: The SELECT statements must appear in the output unaltered.

For exercise solutions, see Appendix A.

7. Model Programming - Defining Branches and Constructs

In COBOL, conditional processing is specified by means of IF (implying OR and AND) statements. They are often used in conjunction with NEXT SENTENCE, ELSE, GO TO, and PERFORM. The GO TO and PERFORM functions direct the execution of the program to transfer control to logical destinations called procedure-names. In this manner, the program is able to select, bypass, or repeat logical steps.

Structured programming constructs may also be available to the COBOL programmer. For example, CA-MetaCOBOL+ Structured Programming constructs IF-ELSE-ENDIF, SELECT-WHEN-ENDSEL, and LOOP-UNTIL-ENDLOOP make possible the preferred one-entry/one-exit condition testing, selection, and repetition.

Similar features are available to the CA-MetaCOBOL+ macro writer. Steps can be selected, repeated, or bypassed either conditionally or unconditionally. Logical destinations can be defined within the same, or even in another macro and accessed by branch-and-return directives.

Up to this point, translation has occurred in a linear fashion (i.e., the macro model has been executed sequentially until the next macro definition). Any attempt to flowchart a macro, thus far, would result in a vertical list of blocks.

Beginning with this section, the macro language takes on the characteristics of a true programming language, with logical decision-making capabilities.

This section describes:

- Leaving the macro using &GOBACK.
- Logical destinations.
- The branch and return directive &DO.
- How to set conditions.
- A definition of constructs.

- Selecting between logical paths using an &IF construct.
- Using &SELECT to choose among logical paths based on the value of one subject or on several conditions.
- Defining a looping structure using &REPEAT.

7.1 Leaving a Macro

The &GOBACK directive allows you to terminate a macro.

Format:

```
&GOBACK
```

&GOBACK
unconditionally terminates the macro in which it appears and returns control to the Translator. It functions like the COBOL GOBACK statement.

Example

```
WP ABC   :
  . . .
  &GOBACK
  . . .
```

7.2 Logical Destinations

Logical destinations are special words, embedded within a macro model, that are used to define locations within the model. These words can be used within certain directive expressions to represent where to transfer control.

Format:

&T*name*

&L*name*

&Tname
defines a tag which can be referenced *only in the macro model where it is defined.*

&Lname

defines a label, or logical destination, which can be referenced *in another macro model within the region.*

Logical destinations may be a maximum of 30 contiguous alphanumeric characters, including the leading characters (&T or &L). They may not contain embedded spaces, punctuation, or quotation marks. (See the "Word" definition in the Glossary.) Please note:

- &Tname must be unique within a macro model and is known only within the macro model where it is defined, so that identical tags can be defined in different macro models.
- &Lname is known to all macros loaded to the region in which it is defined and must be unique within the entire region. The label &LOCAL cannot be used.

Example

In the following macros, the tag &T-SUB-IN defines respective logical destinations within the macros ABC and DEF. The label &L-ERROR is defined in macro ABC; &L-ERROR cannot also be defined in macro DEF, because it is known in both macros ABC and DEF:

```
WP ABC    :
   &T-SUB-IN .  .  .
   .  .  .
   &L-ERROR .  .  .
   .  .  .
WP DEF   :
   &T-SUB-IN .  .  .
   .  .  .
```

7.3 Branch and Return

The &DO directive, like the COBOL PERFORM verb, is a convenient method for sharing macro model code between two or more macros. It causes a transfer of control to a logical destination and a subsequent return when &EXIT is reached.

The &DO directive expression can be specified in the model of a String macro ONLY.

Format:

```
&DO  {&Tname}
     {&Lname}
```

&DO

Execution of the *&DO* directive causes the model to transfer control to the tag or label specified. Execution continues until an &EXIT directive is found. Control then returns to the model statement following the &DO.

&Tname

must be defined in the macro model containing the &DO directive. *&Lname* may be defined within any String macro model in the region, including the one which contains the &DO.

An &DO directive can appear within the code executed by another &DO. Such nesting can continue for a maximum of 32 levels.

```
&EXIT
```

Format:

&EXIT

The *&EXIT* directive is specified in the model of a String macro only and causes return to the model statement following an outstanding &DO directive. If control did not come from an &DO, the &EXIT directive is ignored so that control "falls through" to the model statement following the &EXIT.

Data Item Attribute Example

The following short form macros share model statements to generate COMPUTATIONAL or COMPUTATIONAL-3 data item attribute clauses. In this example:

1. The PVC macro generates the PICTURE and VALUE clauses.

   PVC ignores the &EXIT directive and outputs COMPUTATIONAL.

2. The PVC3 macro transfers control to the PVC model via the &DO directive expression.

   PVC3 generates the PICTURE and VALUE clauses, executes the &EXIT directive, and returns.

3. The model word following &DO outputs COMPUTATIONAL-3.

CA-MetaCOBOL+ Input:

```
SD     PVC &1 &2(L)    :
       &LPVC
           PICTURE &1
           VALUE &2
           &EXIT
           COMPUTATIONAL
SD     PVC3 &1 &2(L)    :
           &DO &LPVC
           COMPUTATIONAL-3
 .  .  .
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  A PVC S9(4) +0.
 77  B PVC3 S9(5) +1.
 .  .  .
```

CA-MetaCOBOL+ Output:

```
 .  .  .
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  A PICTURE S9(4) VALUE +0 COMPUTATIONAL.
 77  B PICTURE S9(5) VALUE +1 COMPUTATIONAL-3.
 .  .  .
```

7.4 Defining Conditions

Several macro model directives select and repeat steps based on the condition of source or macro data. Defining conditional expressions allows the macro programmer to make such tests. This section describes how to define the following *simple conditions* - relation conditions, state conditions, and Boolean conditions. It then shows how they can be combined.

Relation Conditions

The relation condition compares the current values of two items.

Format:

```
subject  relational  object operator
```

subject

The subject of a relation condition can be any of:

Variable
Symbolic Operand
Concatenation
Literal
Symbolic Operand Attribute (see Section 8)

object

The object of the relation condition can be any of these, plus NULL.

Relational operators are:

Relational Operator	Meaning
< or LT	less than
LE	less than or equal to
= or EQ	equal to
NE	not equal to
> or GT	greater than
GE	greater than or equal to

Any relational operator may be negated by preceding it with the word NOT (for example, NOT LT).

Please note:

- The subject can be tested for an empty alphanumeric variable by comparing for equal or not equal to NULL.
- If a symbolic operand is specified, only the first word of its current value is used in the comparison.
- If both subject and object are numeric, a numeric comparison is done; otherwise, a non- numeric comparison is done.
- If an item is a non-numeric literal or a symbolic operand with a non-numeric literal value, bounding quotes are ignored in the comparison.
- If the items are not the same length, the shorter one is expanded with trailing spaces or leading zeros, as appropriate to the class of the comparison.

Example

The following are valid, simple relational conditions:

```
&VNAME EQ NULL
&1 NE `.'
&VCOUNT NOT = 0
```

State Conditions

ENDSCAN is a keyword that can be specified in both simple and combined conditional expressions to determine the "at end" state of an &SCAN-type directive. NOT ENDSCAN is the false state. See Section 8.3, "Retrieving Data Items."

Boolean Conditions

The true (&Bname) or false (NOT &Bname) state of a Boolean variable can be determined by the following conditional expression.

Format:

```
[NOT]  &Bname
```

&Bname
: The condition is expressed simply as the Boolean variable (e.g., &B-FIRST-TIME) or as the false state (e.g., NOT &B-FIRST-TIME).

Example

```
&LOCAL &B-FIRST-TIME = TRUE
       .  .  .
&IF &B-FIRST-TIME
   &SET &B-FIRST-TIME = FALSE
       .  .  .
&ENDIF
```

Combined Conditions

You can combine the simple conditions that were just described with the logical connectives &AND and &OR.

```
simple-condition  {&OR simple-condition  .  .  .  }
                  {&AND simple-condition  .  .  .  }
```

simple-condition
can be a relation, state, or Boolean condition.

&OR
is a directive that "separates" sets of simple conditions.

&AND
is a directive that "connects" simple conditions.

Do NOT intermix &ORs and &ANDs in a single combined condition.

Example

```
&V-1 EQ `S' &OR &V-1 EQ `V' &OR &V-1 EQ `A'

&B-EDIT-OK &AND NOT &B-PREVIOUS-ERRORS

&B-PREVIOUS-WORD-TO &AND &V-ITER NE 1
```

7.5 Constructs

Constructs are combinations of directives within macro models that define the logic and scope of selection and repetition functions.

These control structures are consistent with modern structured programming practice; they ensure a single entry and single exit for each logical function.

In all cases, these constructs pair an explicit beginning directive to an explicit ending directive. For example, the following begin and end an &IF construct:

```
&IF
  .
  .
  .
&ENDIF
```

Additional directives and directive expressions in the construct syntax may define additional processes, such as alternate logical paths. For example, &ELSE defines the end of the true path and the beginning of the false path of an &IF construct.

Additional constructs, other directives, and substitution words can be placed within the scope of a construct, and will be acted upon when the logical path of the construct is executed.

This section describes the following:

- Selection Constructs - used to choose between two (&IF) and multiple (&SELECT) alternate actions.

- A Repetition Construct - used to define a loop and to test for ending conditions (&REPEAT).

7.6 Basic IF Selection

Use the &IF construct in any macro model to select between two logical paths, based on whether a condition is true.

Format:

```
 &IF condition
      process-1
[&ELSE
      process-2]
 &ENDIF
```

&IF
identifies the physical and logical beginning of the &IF construct. *&ENDIF* identifies the physical and logical end of the construct.

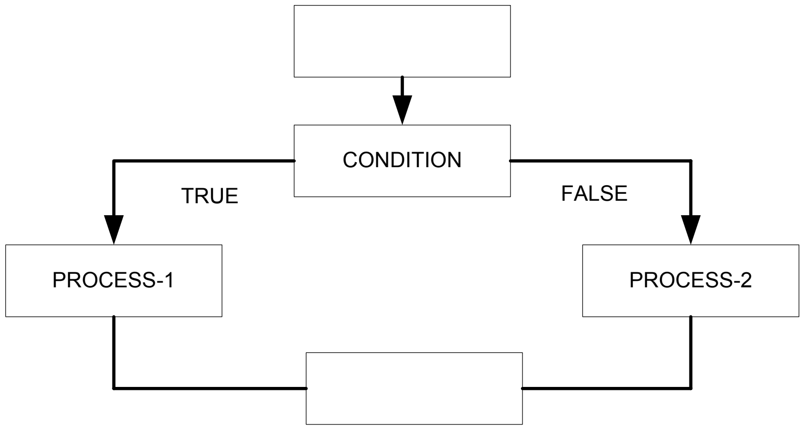
&ENDIF
provides both clarity to the program, highlighting the selection logic, and the capability to nest &IF constructs. It terminates only the most recently unterminated &IF construct. Each &IF must have a matching &ENDIF.

condition
is a simple or combined condition.

process-1 and **process-2**
can be one or more CA-MetaCOBOL+ directives, substitution words, or additional constructs. Either process can be omitted.

&ELSE
is a directive that identifies the end of the true path and the beginning of the false path as a result of the truth test(s) in the condition. If the condition is true, processing continues with process-1. If the condition is not true, processing continues with process-2.

The following flow chart represents the logic of the &IF construct:



&IF-&ELSE-&ENDIF Construct

Save SELECT File Names Example

The previous section contains an exercise in which up to nine file names following Environment Division SELECTs are stored in a global table for subsequent reference by other macros. What happens if there are more than nine SELECTs? The macro might be rewritten to display a diagnostic under this condition, as follows:

CA-MetaCOBOL+ Input:

```
           SE      SELECT &1    :
                   SELECT &1
                   &GLOBAL &VFILE(9) X(30)        /* FILE-NAME
TABLE
                   &LOCAL &VX 99                  /* INDEX
                   &SET &VX = &VX + 1
                   &IF &VX > 9                    /* IF INDEX > 9
                    &NOTE &( `FILE NAME ' &1 ' NOT INCLUDED' &)
                   &ELSE                          /* IF INDEX <= 9
                    &SET &VFILE(&VX) = &1              &ENDIF
           /* END CONSTRUCT
```

7.7 Selection By Current Value/Attributes

Use the &SELECT construct in any macro model to select among any number of logical paths based on the current value or attributes of one subject.

Optional postscripts specify actions to take when any of the comparisons match or none match. This format is similar to the classic structured CASE statement.

Format:

```
 &SELECT subject
 &WHEN object [&OR object] . . .
            process
[&WHEN object [&OR object] . . .
            process] . . .
[&WHEN OTHER
            process]
[&WHEN ANY
            process]
 &ENDSEL
```

&SELECT
identifies the physical and logical start of the &SELECT construct.

&ENDSEL
is a directive that terminates the construct.

The subject can be a symbolic operand (&1 - &15), alphanumeric or numeric variable (e.g., &VNAME), or symbolic operand attribute (e.g., &1'T). (Attributes are discussed in Chapter 8.)

&WHEN
identifies an object to be tested and defines the beginning of a logical path that will be executed when an object is equal to the subject. It also marks the end of the previous &WHEN.

object
can be a symbolic operand, alphanumeric or numeric variable, or attribute, like the subject. The object can also be a literal, concatenation, or NULL. (NULL represents "no value"; i.e., an empty variable.) The subject and object must be of the same class (numeric or alphanumeric).

A set of objects can be connected by the directives &OR or &OR NOT. For example:

```
&WHEN  `V'  &OR  `S'  &OR  NOT  `T'
```

process

Each process can be one or more directives, constructs, substitution words, or can be omitted.

&WHEN OTHER and **&WHEN ANY**

The construct can include the postscripts &WHEN OTHER and &WHEN ANY. OTHER specifies a process to execute only if no condition is true. ANY specifies a process to execute if one of the conditions is true.

This format of the &SELECT works as follows:

1. The process of the first object equal to the subject is executed. Control passes either to the &WHEN ANY process, if specified, or to the next executable model statement following the &ENDSEL.

2. If no object is equal to the subject and if the &WHEN OTHER process is specified, the &WHEN OTHER process is executed. If there is no match and the &WHEN OTHER is not specified, control passes to the next executable model statement after &ENDSEL.

Example

The following illustrates the format of an &SELECT with a single subject.

```
SELECT &V@VERB
&WHEN `MOVE'
      &DO &T-MOVE
&WHEN `ADD'
      &DO &T-ADD
&WHEN `PERFORM'
      &DO &T-PERFORM
&WHEN OTHER
      &DO &T-ERROR
&ENDSEL
```

7.8 Unrestricted Conditional Selection

The &SELECT construct can also have multiple condition tests. Use this form of the construct in a macro model to select among any number of logical paths based on which condition is true. It is a more general version of &SELECT with a single subject. Optional postscripts specify actions to take when any of the tests are true or all are false.

Format:

```
 &SELECT
 &WHEN condition
     process
[&WHEN condition
     process] ...
[&WHEN OTHER
     process]
[&WHEN ANY
     process]
 &ENDSEL
```

&SELECT
The &SELECT directive identifies the physical and logical start of the &SELECT construct.

&ENDSEL
terminates the &SELECT construct.

&WHEN
defines the end of one logical path and the beginning of the next. Each logical path has a corresponding condition test, and that path is executed if the condition is true.

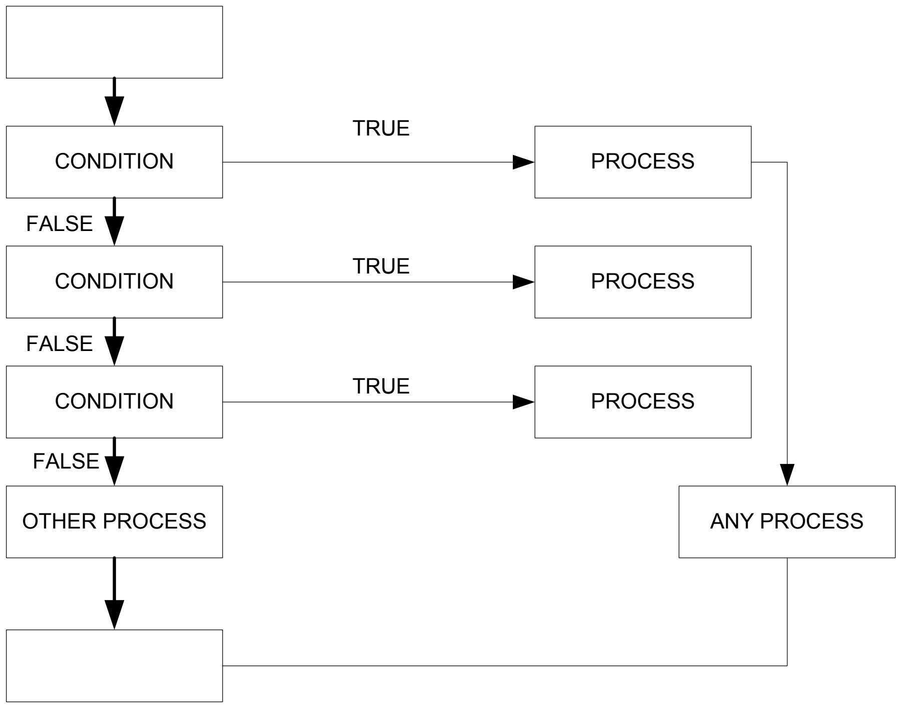
condition
Each condition can be a simple or combined condition.

process
Each process can be one or more directives, constructs, substitution words, or can be omitted.

This form of &SELECT works as follows:

1. The first process whose condition is true is executed, and control passes either to the &WHEN ANY process, if specified, or to the next executable model statement following the &ENDSEL.

2. If none of the conditions are true and the &WHEN OTHER postscript is specified, the &WHEN OTHER process is executed. If none of the conditions are true and the &WHEN OTHER postscript is not specified, control passes to the next executable model statement following &ENDSEL.



&SELECT-&WHEN-&ENDSEL Construct

Example

The following example illustrates the format for an &SELECT with three conditions, an ANY postscript, and an OTHER postscript.

```
&SELECT
&WHEN &1 = `END'
      .  .  .
&WHEN &V@VERB = `MOVE'
      .  .  .
&WHEN &3 = &V@KEYWORD
      .  .  .
&WHEN ANY
      .  .  .
&WHEN OTHER
      .  .  .
&ENDSEL
```

7.9 Repetition

Use the &REPEAT construct to set up an inline, logical loop. The construct allows you to define a condition that can be tested and that will terminate the loop before any processing, after all processing, or any time between.

Format:

```
&REPEAT
[process]
[ &UNTIL condition ]
[    [process]      ]...
&ENDREP
```

&REPEAT
identifies the physical and logical beginning of the loop structure. &ENDREP is a directive that terminates an &REPEAT loop.

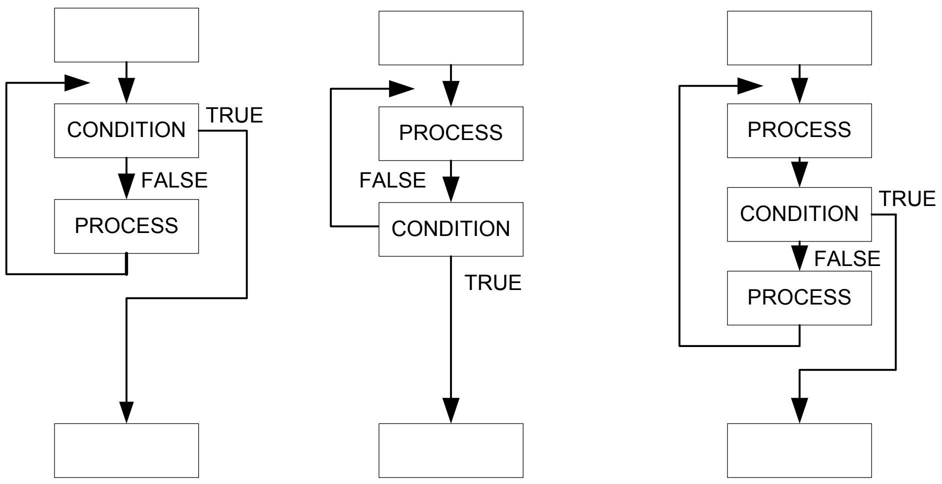
process
Each process is one or more directives, constructs, substitution words, or can be omitted.

&UNTIL
tests the condition and terminates the &REPEAT construct when the condition is true. The condition can be a simple or combined condition. (More than one &UNTIL can be used when a test requires both &AND and &OR directives.)

Using &REPEAT:

1. Execution proceeds until the condition is tested.
2. If the condition is true, control passes to the next executable model statement following the &ENDREP.
3. If the condition is not true, processing continues from the test through the end of the loop, and the next iteration begins.
4. This constitutes a "½ or n+½" times loop and is repeated until the condition is true.

As the following diagram shows, you can place the condition test first in the loop, last in the loop, or anywhere in between.



&REPEAT-&UNTIL-&ENDREP Construct

Escaping from a Loop

To terminate an &REPEAT loop before the condition test is true, use the &ESCAPE directive.

Format:

&ESCAPE

&ESCAPE

When an &ESCAPE directive is executed, it causes the &REPEAT loop in which it occurs to terminate unconditionally. Control passes directly to the next executable statement after the corresponding &ENDREP.

Generate COBOL READ Example

The following macros generate COBOL READ statements. The format of the READ depends on whether the associated file access is sequential or relative. These macros determine the organization of the file using the access defined in the ASSIGN clause of the SELECT statement in the File-Control Section.

- The SELECT macro acquires each COBOL SELECT and ASSIGN and determines the organization of the file. Files are defined as one of: Physical sequential (S), VSAM sequential (AS), or VSAM random (R). An &DO calls the routine labelled &L@PUT-GET-FILENAME to put the file-name and its organization in a table.

- The READ macro calls the same routine to get a file-name and organization. It outputs the corresponding READ statement. For sequential files, it uses an AT END clause; for random files an INVALID KEY clause.

- Macro $$SUB$$ will not match any input source. It is called to build a table with up to nine entries. It stores or retrieves using the file-name and organization that has been passed. It uses an &REPEAT to loop through the table. The nested &WHEN paths specify optional processes when the file-name is in the table, when it is not in the table, or when the tenth file-name has been passed.

When any of these have occurred, the macro escapes from the loop. When none have occurred, it increments a counter and continues the loop.

CA-MetaCOBOL+ Input:

```
SE     SELECT &1 ASSIGN TO &2 :
       &GLOBAL &V@ORG = NULL X(8)             /* ORGANIZATION
       &GLOBAL &V@DDN = NULL X(8)             /* WORK AREA
       &GLOBAL &V@X   = 0    99               /* INDEX INTO &2
       &GLOBAL &V@C   = NULL X                /* &2 CHARACTER
       &SET &V@ORG = NULL
       &SET &V@DDN = NULL
       &SET &V@X = 0
       &SET &V@C = NULL
       &REPEAT                                /* LOOP THRU &2
            &SET &V@X = &V@X + 1              /* ONE CHARACTER
            &SET &V@C = &2 % &V@X             /* AT A TIME
       &UNTIL &V@C EQ NULL                    /* UNTIL NULL FOUND
            &IF &V@C EQ `-'                   /*  IF END OF ORG.
                 &SET &V@ORG = &V@DDN         /*  SAVE FORMAT
                 &SET &V@DDN = NULL
            &ELSE                             /*  OTHERWISE
                 &SET &V@DDN = &( &V@DDN &V@C &) /*  BUILD ORG.
            &ENDIF
       &ENDREP                                /* END LOOP
       &IF &V@ORG NE `S' &AND &V@ORG NE `AS'  /* IF ORG.  NE S/AS
            &SET &V@ORG = `R'                 /*  ORG.  = R
       &ENDIF
       &DO &L@PUT-GET-FILENAME                /* DO PUT-GET
S      $$SUB$$ :
 &L@PUT-GET-FILENAME
       &GLOBAL &V@FNAME(10) = NULL X(30)      /* FILE TABLE
       &GLOBAL &V@FORG(10)  = NULL X(2)       /* ORGANIZATION
       &GLOBAL &V@FX        = 0    99         /* INDEX
       &SET &V@FX = 1
       &REPEAT                                /* LOOP THRU TABLE
            &SELECT
            &WHEN &1 EQ &V@FNAME(&V@FX)       /* FILE IN TABLE:
                 &SET &V@ORG = &V@FORG(&V@FX) /*  GET ORG.
            &WHEN &V@FNAME(&V@FX) EQ NULL     /* FILE NOT IN TABLE:
                &SET &V@FNAME(&V@FX) = &1     /*  PUT FILE NAME
                &SET &V@FORG(&V@FX) = &V@ORG  /*  & ORG.
            &WHEN &V@FX EQ 10                 /* 10TH FILE:
                &NOTE `TOO MANY FILE NAMES'   /*  NOTE
            &WHEN ANY                         /* GET/PUT/NOTE:
                &ESCAPE                       /*  END LOOP
            &WHEN OTHER                       /* NO GET/PUT/NOTE:
                &SET &V@FX = &V@FX + 1        /* REPEAT
            &ENDSEL
        &ENDREP
 &EXIT
SP      READ &1:
        &DO &L@PUT-GET-FILENAME
        &IF &V@ORG EQ `R;
            READ &1 INVALID KEY GO TO &( INV- &1 &)
        &ELSE
            READ &1 AT END GO TO &( EOF- &1 &)
        &ENDIF
$ED
        SELECT FILE1 ASSIGN TO S-DDN1.
        SELECT FILE2 ASSIGN TO AS-DDN2.
        SELECT FILE3 ASSIGN TO DDN3.
        SELECT FILE4 ASSIGN TO DDN4.
        SELECT FILE5 ASSIGN TO S-DDN5.
        SELECT FILE6 ASSIGN TO AS-DDN6.
        SELECT FILE7 ASSIGN TO DDN7.
        SELECT FILE8 ASSIGN TO DDN8.
        SELECT FILE9 ASSIGN TO AS-DDN9.
        SELECT FILE10 ASSIGN TO DDN0.
        SELECT FILEA ASSIGN TO DDNA.
****    NOTE N99  `TOO MANY FILE NAMES'
$DD
.  .  .
$PD
        READ FILE1.
        READ FILE2.
        READ FILE3.
        READ FILE4.
        READ FILE5.
        READ FILE6.
        .  .  .
```

CA-MetaCOBOL+ Output:

```
ENVIRONMENT DIVISION.
.  .  .
DATA DIVISION.
.  .  .
PROCEDURE DIVISION.
      READ FILE1
        AT END
        GO TO EOF-FILE1.
      READ FILE2
        AT END
        GO TO EOF-FILE2.
      READ FILE3
        INVALID KEY
        GO TO INV-FILE3.
      READ FILE4
        INVALID KEY
        GO TO INV-FILE4.
      READ FILE5
        AT END
        GO TO EOF-FILE5.
      READ FILE6
        AT END
        GO TO EOF-FILE6.
      .  .  .
```

7.10 Summary - Defining Branches and Constructs

Listed below are the key points covered in this section.

- Use &GOBACK to transfer control to the end of the macro.
- You can define logical destinations in macro models. Tags (&Tname) are local to a macro; labels (&Lname) can be shared among String macros.
- &DO-&EXIT defines a branch to a tag or label and the subsequent return; it can be used in String macros only.
- Conditional expressions allow execution options based on program tests; they can be:

Relational	- e.g., &VNUM = `123'
State	- ENDSCAN and NOT ENDSCAN
Boolean	- e.g., &B-FLAG or NOT &B-FLAG
Combined	- simple conditions joined by either &OR or &AND (not mixed)

- Selection and repetition constructs have a single entry and a single exit; they can be nested.
- &IF-&ELSE-&ENDIF selects one of two paths based on a single condition.
- &SELECT-&WHEN-&ENDSEL with multiple conditions selects among any number of paths, each path having its own condition.
- &SELECT-&WHEN-&ENDSEL with a single subject selects among any number of paths based on the current value of the subject.
- &REPEAT-&UNTIL-&ENDREP repeats a loop with a condition test at the beginning, at the end, or in the midst of the loop.
- &ESCAPE terminates an &REPEAT loop.

Exercise 7: Given an indexed global variable defined as,

```
&GLOBAL &VFILE(9) X(30)
```

and containing up to nine file names, write a Word macro that replaces CLOSEALL with:

```
CLOSE file-name-1 file-name-2 .  .  .
```

CA-MetaCOBOL+ input might appear as:

```
.  .  .
ENVIRONMENT DIVISION.
.  .  .
FILE-CONTROL.
  SELECT INPUT-FILE  ASSIGN TO .  .  .
  SELECT MASTER-FILE ASSIGN TO .  .  .
  SELECT REPORT-FILE ASSIGN TO .  .  .
  .  .  .
PROCEDURE DIVISION.
.  .  .
  CLOSEALL.
  .  .  .
```

Procedure Division translation should be:

```
.  .  .
PROCEDURE DIVISION.
.  .  .
   CLOSE INPUT-FILE MASTER-FILE REPORT-FILE.
   .  .  .
```

Remember to reinitialize the required variables, in case more than one CLOSEALL is specified. Do not generate a CLOSE statement if the table is empty.

For exercise solutions, see Appendix A.

8. Model Programming - Acquiring Macro Model Input

Previous sections have shown how symbolic operands are used to acquire a word or words that match macro prototypes. Thus far, words have been acquired solely from input source. In this section, macro model programming is extended to acquire additional information for the macro model.

This section describes:

- How to acquire additional attributes of a symbolic operand's current value
- How to set and modify the current value of a symbolic operand using &EQU
- How to retrieve information from the Data Division.

8.1 Acquiring Symbolic-Operand Attributes

Symbolic-operand attributes are additional characteristics of the current value of a symbolic operand, such as its type and size. The CA-MetaCOBOL+ macro model can access this information.

A similar facility is programmed into the various COBOL compilers. For example, when a data item is referenced by name in the Procedure Division, the COBOL compiler can associate it to the data description to determine location, size, usage, etc., and then build the proper machine instructions.

The notation for representing attributes of a symbolic operand takes the form:

Format:

```
        &n'attribute
```

&n

can be any symbolic operand &1 through &15 or &0. (&0 is described later under Retrieving Data Items.)

A single quote (') without surrounding spaces is required.

attribute

is a 1-character code which refers to a specific characteristic of the current value of the symbolic operand. The valid attribute codes are listed with the first example following. Complete descriptions appear in Appendix C.

For example, the size in bytes of the current value of a data item stored in &5 is represented as:

```
                &5'S
```

Some attributes apply only to data items (an entry defined in the Data Division). Such attributes are known as *data item attributes*.

A data item attribute can be referenced only when the associated data item has been completely defined. In general, this means that you can access the attribute from a macro called in the Procedure Division (that is, a macro with a P division code or one that will only match Procedure Division input source). However, the attribute may be available to a Data Division macro after the associated data item is completely defined. (A group item is not completely defined until another group item begins.)

Other symbolic operand attributes refer to literals, figurative constants, and other current values of symbolic operands, and can be accessed in macros called in any COBOL division.

Attribute Example

Assume that during CA-MetaCOBOL+ tranlsation, a macro is being executed which contains the symbolic operand &1 (representing the data-name TAX-RATE). The record definition containing this item shows the following:

```
             01 TAXES.
                02 FILLER            PIC X(7).
                02 TAX-BLOCK         USAGE IS COMP-3.
                   03 FILLER         PIC S9(2)V9(3) VALUE +0.750.
                   03 FILLER         PIC S9(2)V9(3) VALUE +1.250.
                   03 FILLER         PIC S9(2)V9(3) VALUE +1.625.
                02 TAX-TABLE         REDEFINES TAX-BLOCK.
                   03 TAX-RATE       OCCURS 3 TIMES PIC S9(2)V9(3)
COMP-3.
```

The attributes of &1 (TAX-RATE) that are available for analysis by the CA-MetaCOBOL+ macro include:

	Attribute	**Content**	**Explanation**
&1'D	(Displacement)	7	Location relative to beginning of record: TAX + (97) FILLER.
&1'9	(Display Size)	5	Number of positions needed to display contents of TAX-RATE.
&1'L	(Level)	\`03'	Level number of TAX-RATE.
&1'N	(Name Size)	8	Number of characters in first word (TAX-RATE).
&1'0	(OCCURS)	3	Maximum value of OCCURS clause (3 times).
&1'K	(OCCURS Level)	1	Number of subscripts (1).
&1'P	(Point)	3	Decimal point location (3 decimal positions).
&1'R	(REDEFINES)	1	Associated with REDEFINES (0 = not associated with REDEFINES).
&1'-	(Sign)	1	PIC is signed (0 = not signed).
&1'S	(Size)	3	3 bytes define the data item.
&1'Y	(SYNCHRONIZED)	0	Data item not SYNCHRONIZED (1 = SYNCHRONIZED).
&1'T	(Type)	\` '	Type is space because TAX-RATE is a data-name. It is not: -an Area A indicator (\`A') -a literal or figurative constant (\`L') -a NOTE or comment (\`N') -a match to a String macro name (\`S') -a verb or terminating period (\`V')
&1'U	(USAGE)	\`3'	COMP-3.
&1'V	(VALUE)	0	No VALUE clause (1 = VALUE clause).

Maximum Table Entries Example

The following example shows how to use the OCCURS attribute. A String macro is defined to substitute the maximum number of table entries in place of the words "MAX data-name". If the program is maintained at the CA-MetaCOBOL+ source level, the macro substitutes the proper value automatically, even if the number of entries in the table is changed:

CA-MetaCOBOL+ Input:

```
           SP      MAX &1(Q)   :
                       &1'O
            .  .  .
            DATA DIVISION.
            WORKING-STORAGE SECTION.
            01 TABLE.
               02 TABLE-ENTRIES     OCCURS 200 TIMES INDEXED BY
X1
                  PICTURE X(80).
            .  .  .
            PROCEDURE DIVISION.
            .  .  .
                   SET X1 UP BY 1.
                   IF X1 IS GREATER THAN MAX TABLE-ENTRIES ...
            .  .  .
```

CA-MetaCOBOL+ Output:

```
            .  .  .
            DATA DIVISION.
            WORKING-STORAGE SECTION.
            01 TABLE.
               02 TABLE-ENTRIES     OCCURS 200 TIMES INDEXED BY
X1
                  PICTURE X(80).
            .  .  .
            PROCEDURE DIVISION.
            .  .  .
                   SET X1 UP BY 1.
                   IF X1 IS GREATER THAN 200 ...
            .  .  .
```

Exercise 8: Write a macro to interrogate each MOVE statement in a COBOL program. If the receiving field of a MOVE statement contains more than 256 bytes, print the diagnostic,

data-name IS LARGER THAN 256 BYTES

For exercise solutions, see Appendix A.

8.2 Modifying Symbolic Operands

In general, a symbolic operand's value is acquired from the input source, while a variable's value is defined in the macro model. However, the current value of a symbolic operand can also be defined or modified in macro models by the &EQU directive expression. Applications include:

- Alternate forms of a macro can share logical model statements by equating symbolic operands.
- Data-names passed as the current values of variables can be placed in symbolic operands for attribute analysis.
- Symbolic operands can store and pass multiple words and retain quotes on alphanumeric literals (often in conjunction with symbolic operand storage variables).
- Symbolic operands can be modified to contain additional words for analysis, etc.

To set the current value of a symbolic operand, use the &EQU directive expression. This directive expression is similar to an &SET, except that a symbolic operand is modified, not a variable.

Format:

```
&EQU &n sending-item
```

&EQU
sets &n to the value of sending-item.

&n
is a symbolic operand (&1 through &15 - not &0).

sending-item
can be a symbolic operand (&0 through &15), a variable, a concatenation, a literal, or an attribute.

The sending-item can be a symbolic operand storage variable. The &EQU directive is the only means for placing the current value of a symbolic operand storage variable in a symbolic operand. (See also the *Macro Facility Reference*).

Symbolic Operand Concatenation

A special form of concatenation is reserved for use by the sending item of &EQU. The directive &(E begins the symbolic operand concatentation. &) ends the concatenation. The expression can contain constants, variables, and symbolic operands separated by one or more blanks.

Unlike the other forms of concatenation, the integrity of each word in the expression is retained in &n. (The other forms of concatenation can also be specified in the sending item of the &EQU directive.)

For example, if &1 contains the current value EMPLOYEE-NUMBER, the following places EMPLOYEE-NUMBER OF MASTER in &2:

```
          &EQU &2 &(E &1 OF MASTER &)
```

Table-handling Efficiency Example

Table-handling in COBOL is extremely inefficient if subscripts are defined as packed (COMPUTATIONAL-3) or external decimal (DISPLAY). Binary subscripts (COMPUTATIONAL), indexing, and absolute addressing by means of a numeric literal are more efficient. Also, each level of subscripting or indexing compounds inefficiency. Since table-handling identifiers can be specified as operands of many COBOL statements, checking for table-handling efficiency can require a great deal of complex logic within many macro models.

The problem can be simplified by the following example. Two macros perform these functions:

1. A String macro acquires these identifiers.

2. A Word macro, defined as @SS@ so that it will not match source words, is called by the String macro to analyze each identifier and display a diagnostic where appropriate. The symbolic operands &15 and &14 are used to pass the appropriate identifiers to @SS@ and to access the attributes of any subscripts or indices, respectively.

CA-MetaCOBOL+ Input:

```
  SP     MOVE &1(S,L) TO &2(S)   :
         MOVE &1 TO &2
         &IF &1'K GT 0                          /* IF SUBSCRIPTED,
             &EQU &15 &1                        /*  SAVE IN &15 AND
             @SS@                               /*  CALL MACRO @SS@
         &ENDIF
         &IF &2'K GT 0
             &EQU &15 &2
             @SS@
         &ENDIF
```

```
 WP     @SS@ :
        &LOCAL &V@WORD X(30)                /* WORD WORK AREA
        &LOCAL &V@INDEX 99                  /* INDEX
        &SET &V@INDEX = 0
        &IF &15'K GT 1                      /* >1 SUBSCRIPT?
            &NOTE &( &15 ' - TOO MANY SUBSCRIPTS.' &) /* YES
        &ENDIF
        &REPEAT                             /* LOOP TO (
            &SET &V@INDEX = &V@INDEX + 1
            &SET &V@WORD = &15 # &V@INDEX
        &UNTIL &V@WORD EQ '(' &OR
            &V@WORD EQ NULL
        &ENDREP
        &REPEAT                             /* LOOP THRU
            &SET &V@INDEX = &V@INDEX + 1    /*  SUBSCRIPTS,
            &SET &V@WORD = &15 # &V@INDEX   /*  NOTE IF USAGE
        &UNTIL &V@WORD EQ ')' &OR           /*  EQ COMP-3 ('3')
            &V@WORD EQ NULL                 /*  OR DISPLAY ('9')
            &EQU &14 &V@WORD
            &IF &14'U EQ '3' &OR $14'U EQ '9'
                &NOTE &( &V@WORD ' - PACKED/DISPLAY SUBSCRIPT.' &)
            &ENDIF
        &ENDREP
  .  .  .
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  77    SS3 PIC S9(4).
  77    WORK-AREA PIC S9(5).
  01    TABLE-AREA.
        02 DEPARTMENT OCCURS 10 TIMES INDEXED BY X1.
             03 DEPT-CODE PIC 9(3).
             03 DEPT-NAME PIC X(20).
             03 DEPT-RATE OCCURS 5 TIMES PIC S9(5).

  .  .  .
  PROCEDURE DIVISION.
  .  .  .
        MOVE SPACE TO DEPARTMENT (X1).
        MOVE DEPT-RATE (1, SS3) TO WORK-AREA.
****NOTE N99 DEPT-RATE - TOO MANY SUBSCRIPTS.
****NOTE N99 SS3 - PACKED/DISPLAY SUBSCRIPT.
        GOBACK
```

8.3 Retrieving Data Items

Macro models in the Procedure Division often need to retrieve the COBOL data-names that have been generated in the Data Division. For example, given a record-name, it might be necessary to determine the file-name or, conversely, the data-names and to acquire attributes of subordinate items.

CA-MetaCOBOL+ allows the macro model to scan the Data Division that has been read and stored and to retrieve the required data definitions. A special symbolic operand &0 is used. Two ways to retrieve data items are described here:

- *Retrieving sub-items* - determines data-names in the order in which they are defined.

- *Retrieving group items* - determines the name of the next higher item in the data hierarchy.

Other directives retrieve data-names for a Data Structure Table (TDS) relative address (&SCANA), level-88 condition-names (&SCANC), or index-names (&SCANI). See the *Macro Facility Reference* for details.

You cannot retrieve a data-name until it is completely defined. In general, this means that you can retrieve the data-name from a macro called in the Procedure Division. However, the data-name may be available to a Data Division macro after it is completely defined. (A group item is not completely defined until another group item begins.)

Retrieving Sub-items

The &SCAN directive expression, logically placed within a processing loop in a macro model, can be used to retrieve successive data-names as the current value of symbolic operand &0.

Format:

```
&SCAN start-item end-item
```

&SCAN
Each time **&SCAN** is executed, it retrieves the next data-name within the range.

start-item
is the first data item retrieved.

end-item
is the last data item retrieved.

The data name becomes the current value of symbolic operand &0. The entire function is most easily represented by the COBOL-like statement,

```
          READ from-1/thru-2 INTO &0
```

where each data item is treated as a separate record.

The start-item can be a data-name, file-name, or one of the Data Division section-names - FILE, WORKING-STORAGE, LINKAGE, or COMMUNICATION. The pseudo section name PROCEDURE must be used to access items generated out-of-line to the Data Division while &DSTART is active; see Section 10.3. The start-item can be a symbolic operand (&1 through &15) containing a unique identifier or an alphanumeric variable containing a unique data-name or section-name.

The end-item represents the end of the &SCAN range. It can be any of the start-item types. Please note:

- The end-item must either follow the start-item in the Data Division, or to search for one item, be identical to the start-item.

- If the end-item is a section-name or group item, the range of the &SCAN includes all subordinate data items under the end-item.

- Filler items and level-88 conditionals are not retrieved.

The state condition ENDSCAN indicates when the last data item in the requested range has been retrieved. It is set to True after the &SCAN list is exhausted.

&SCAN retains a pointer to the next data item in the range but does not itself search through the range. To retrieve successive data items, code the &SCAN directive expression within a macro loop. Use the state condition ENDSCAN (or its negation NOT ENDSCAN) to test when the items requested have been exhausted and the loop is to end. The following sequence takes place:

1. Once an &SCAN function has been initiated, it automatically OPENs the range to be examined and places the first data-name in &0.

2. Subsequent executions of the &SCAN directive in the loop place successive data-names from the data hierarchy into &0. &0 can be analyzed each time through the loop, and the data-name can be saved or processed.

3. After the &SCAN list is exhausted, the next execution of &SCAN sets the ENDSCAN condition to True.

An &SCAN function cannot be nested within another &SCAN. However, an &SCAN function can be forced to close prematurely by executing the &SCANX directive in the following format:

Format:

```
        &SCANX
```

&SCANX
Once &SCANX is issued, &0 is no longer available. If &0 is needed after execution of &SCANX, save it in a variable or another symbolic operand before terminating the scan (using &EQU).

Please note:

- Executing an &SCAN directive before a prior &SCAN has exhausted all data items in its range or has been closed by an &SCANX is illegal and results in a diagnostic.

Scan Data Items Example

The following example demonstrates use of the &SCAN directive and conditional testing of attributes. The source statement,

```
             BACKGROUND record-name
```

is to be replaced by,

```
             MOVE SPACES TO record-name
```

Numeric data items within the record, however, should be set to zeros, not spaces. As a result, the record description must be examined for numeric items. Where these are found, the statement "MOVE ZEROS TO data-item" is generated. To simplify the example, OCCURS and REDEFINES attributes are ignored.

In this example:

1. The macro first checks that the data item is defined and valid.

2. If the data item is valid, the macro simply substitutes "MOVE SPACES TO record-name" in place of the "BACKGROUND record-name" statement.

3. The &REPEAT loop scans the record and, for each numeric item, substitutes "MOVE ZEROS TO data-item" statements.

   The first execution of the &SCAN directive expression places MASTER in &0. MASTER is a group item with usage 'G' (not '0', '3', or '9'). The loop, therefore, is repeated.

   The second execution of the &SCAN places MASTER-KEY in &0. Since the usage is '9' (numeric DISPLAY), the statement "MOVE ZEROS TO MASTER-KEY" is generated and the loop is repeated.

The third execution of the &SCAN places MASTER-ID (usage 'X') in &0.

The fourth execution of the &SCAN places MASTER-AMT (usage '3') in &0 and substitutes "MOVE ZEROS TO MASTER-AMT".

The fifth execution of the &SCAN finds the list exhausted, sets ENDSCAN to True, and terminates the loop.

CA-MetaCOBOL+ Input:

```
         SP     BACKGROUND &1(Q) :
                &IF &1'U = 'U'
                    &NOTE &(Q BACKGROUND ' ' &1 ' ' UNDEFINED &)
                &ELSE
                    MOVE SPACES TO &1
                    &REPEAT
                          &SCAN &1 &1
                    &UNTIL ENDSCAN
                          &SELECT &0'U
                          &WHEN '0' &OR '3' &OR '9'
                                 MOVE ZEROS TO &0
                          &ENDSEL
                    &ENDREP
                &ENDIF
          .  .  .
          DATA DIVISION.
          .  .  .
          WORKING-STORAGE SECTION.
          01    MASTER.
                02 MASTER-KEY PIC 9(4).
                02 MASTER-ID PIC X(9).
                02 MASTER-AMT PIC S9(5) COMP-3.
          .  .  .
          PROCEDURE DIVISION.
          .  .  .
                BACKGROUND WORK-AREA.
*********NOTE       N99  'BACKGROUND WORK-AREA UNDEFINED'
                BACKGROUND MASTER.
```

CA-MetaCOBOL+ Output:

```
.   .   .
DATA DIVISION.
.   .   .
WORKING-STORAGE SECTION.
01    MASTER.
      02 MASTER-KEY          PIC 9(4).
      02 MASTER-ID                PIC X(9).
      02 MASTER-AMT          PIC S9(5) COMP-3.
.   .   .
PROCEDURE DIVISION.
      .   .   .
      MOVE SPACES TO MASTER
      MOVE ZEROS TO MASTER-KEY
      MOVE ZEROS TO MASTER-AMT.
.   .   .
```

Retrieving Group Items

To acquire successively higher data-names from the COBOL Data Division, use the &SCANF directive expression in a macro model loop. Each execution of &SCANF returns the names of the next higher item in the data hierarchy as the current value of symbolic operand &0.

Format:

```
&SCANF &n
```

&SCANF
retrieves the name of a single data item - the parent of the data item specified in &n.

&n
must be a symbolic operand from &0 through &15 containing a unique identifier.

Please note:

- If &n refers to a subordinate data item, the name of its next higher group item is returned in &0.
- If &n refers to an index-name, the name of the first item defining the index is returned in &0.
- If &n refers to a condition name, the name of the associated data item is returned in &0.

The highest level of qualification is a record-name in the Working-Storage or Linkage Section or a file-name or code-name in the File or Communication Section, respectively.

The ENDSCAN state condition is set to True when &SCANF attempts to retrieve an item that is not subordinated (and is not an index or condition name).

Unlike &SCAN, &SCANF does NOT maintain a pointer to the next data item. Therefore, to retrieve successive items, each time &SCANF is executed, &n must be updated and set to the previously retrieved data item.

Conversion Example

There are instances when, given an identifier, you must determine where it is defined: Is it in the Working-Storage, File, or Linkage section?

As an example, consider a conversion from a VSE to an MVS COBOL compiler. A common practice under VSE is to read a file and move HIGH-VALUE to various fields on the record on the "AT END" condition. Such logic is illegal under the target compiler if the record is subordinate to the file description, since the input area is no longer addressable. A useful macro displays a warning if a "MOVE HIGH-VALUE TO identifier" statement is found, and "identifier" is defined in the File Section. This can be determined if the highest level of qualification is a file-name (level attribute '00').

Note that the following macro moves the scan to successively higher levels of the data hierarchy each time through the processing loop. In this case, the data item returned in &0 is saved in &2 by an &EQU; and &2 then becomes the operand for the next &SCANF.

CA-MetaCOBOL+ Input Listing:

```
SP      MOVE HIGH-VALUE TO &1(S) :
        MOVE HIGH-VALUE TO &1
        &EQU &2 &1
        &REPEAT
            &SCANF &2
        &UNTIL ENDSCAN
            &IF &0'L = '00'
                  &NOTE
                  &( &1 ' - IS IN BUFFER.' &)
                  &ESCAPE
            &ENDIF
            &EQU &2 &0
        &ENDREP
 . . .
```

```
               DATA DIVISION.
               FILE SECTION.
               FD       MASTER.
               01       M-RECORD.
                        02 M-KEY PIC X(5).
                        . . .
               WORKING-STORAGE SECTION.
               01       WS-RECORD.
                        02 WS-KEY PIC X(5).
                        . . .
               PROCEDURE DIVISION.
               . . .
                        READ OLD-MASTER
                             AT END
                             MOVE HIGH-VALUE TO M-KEY.
********NOTE            N99        M-KEY - IS IN BUFFER.
                         MOVE HIGH-VALUE TO WS-KEY.
                         . . .
```

8.4 Summary - Acquiring Macro Model Input

Listed below are the key points covered in this section.

- The macro model can acquire input words either directly or as the current values of symbolic operands; the macro model can also acquire various attributes of symbolic operands.
- An attribute is specified in the form - &n'attribute - for example &10'A (for a complete list of attribute codes, see Appendix C). Attributes of data items are only available after the data item is completely defined.
- The &EQU directive expression can set a symbolic operand to a literal, a concatenation, the current value of a variable or another symbolic operand, or to a symbolic operand attribute.
- An &(E...&) concatenation can be used with &EQU; it strings together words and the current values of variables and symbolic operands into a new multi-word symbolic operand (i.e., the integrity of each word is preserved).
- A String macro model can retrieve data-names that have been input to the Data Division; the macro can be invoked in the Procedure Division (or in the Data Division at a point after the data item has been completely defined).

  Data items are saved as the current value of symbolic operand &0.

  &SCAN retrieves a range of data-names, one with each execution of &SCAN, in the order in which they are defined.

&SCANF retrieves a single data-name--the parent of a specified data item.

- The state conditional ENDSCAN is set to True when an &SCAN range is exhausted or an &SCANF data item has no parent.

- SCANX is used to leave an &SCAN loop before its range is exhausted.

Exercise 9: An OPEN-PRINT macro was previously defined to OPEN the printer file and then pass the record-name, depth-of-page, and name of the page-heading-routine to the PRINT macro. For ease of reference, this macro is shown below:

```
SP  OPEN-PRINT &1 USING &2 DEPTH &3(L)
                  OVERFLOW &4:
        OPEN OUTPUT &1.
        PERFORM &4.
        &GLOBAL &VRECORD X(30)
        &GLOBAL &VDEPTH 9(2)
        &GLOBAL VOFLO X(30)
        &SET &VRECORD = &2
        &SET &VDEPTH = &3
        &SET &VOFLO = &4
```

The "OPEN-PRINT file-name USING record-name" portion in the prototype was required because the OPEN statement needs the file-name, and the WRITE statement requires the record-name.

Given the file-name, it is relatively easy to find the record-name by looking for the first '01' level entry after the printer FD. Modify the above macro so that the syntax of the OPEN-PRINT macro prototype reads:

```
  OPEN-PRINT &1 DEPTH &2(L) OVERFLOW &3
```

For exercise solutions, see Appendix A.

9. Model Programming - Parsing Input Source

Earlier sections describe how to acquire words from the input source by matching macro prototypes to input source words.

As long as the precise sequence of input words is known beforehand, String macro prototypes can be defined that will match all possible input sequences and invoke the appropriate model statements.

The COBOL language, fortunately, does not always require the programmer to arrange source words in such a precise sequence. For example, in a COBOL data item description, the PICTURE clause can appear before or after the VALUE clause; or, in a MOVE statement, any number of receiving fields can be specified. To control translation under all circumstances would require an impractical number of String macro prototypes or, as this section describes, the ability to examine and obtain each source word under macro control.

This section includes:

- How to use the &GET directive expression to examine the next source word.
- How to use Type attributes to decide whether the word pertains to the current macro model.
- How to use the &STOW and &STORE directives with &GET to remove the word from the input source.

These features provide the CA-MetaCOBOL+ programmer with a powerful tool that can directly control the parsing of input source.

Note: These directive expressions can be executed in the models of String macros only.

9.1 Copying Source Words

The &GET directive expression can be used in a logical loop within a String macro model to copy successive words from the input stream.

Format:

```
&GET &n
```

&GET
copies the next word from the input stream into the specified symbolic operand **&n** (&1 - &15).

- If &n matches a Word or Prefix prototype, the word or prefix is expanded, and the symbolic operand contains the first word of the expansion.
- The &GET directive does NOT remove the source word from the input stream. Thus, the word can be analyzed to determine if it applies to the macro in which the &GET is used, or if it should be left in the input.

9.2 Removing Source Words

The &GET directive expression permits the macro writer to "look ahead" to the next source word. Before subsequent words can be examined by the &GET, however, the current word obtained must be removed from the source. This is the function of the &STORE and &STOW directives.

Format:

```
&STORE

&STOW
```

&STORE
removes the word copied by the last &GET from the input source. This permits the next &GET to copy another word.

&STOW
like &STORE, also removes the word copied by the last &GET executed and permits another execution of the &GET. The &STOW, however, removes the word plus all of its qualifiers and subscripts and saves them in the symbolic operand used in the &GET.

For example, given the statement:

```
            MOVE  A  TO  B  OF  C(X)
```

the following words are acquired each time through an &GET/&STOW loop and an &GET/&STORE loop, respectively:

&GET/&STOW	**&GET/&STORE**
MOVE	MOVE
A	A
TO	TO
B OF C(X)	B
	OF
	C
	(
	X
	)

The &STOW and &STORE directives implicitly reference the symbolic operand obtained by the previous &GET.

Please note:

- The &STORE directive is generally preferred for &GET loops in macros called in the Identification, Environment, and Data Divisions. &STOW is generally required in Procedure Division macros to acquire identifiers and qualified procedure-names.

- Executing successive &GET directive expressions in the same macro call without an intervening &STORE or &STOW is illegal and results in a diagnostic.

MOVE Efficiency Example

Exercise 8 calls for a macro to interrogate each MOVE statement in a COBOL program. If the receiving field of a MOVE statement contains more than 256 bytes, it is diagnosed as inefficient. However, a COBOL MOVE statement can contain any number of receiving fields. Assuming that each MOVE statement is a complete sentence, the following &GET/&STOW loop permits interrogation of each receiving field.

1. The &GET directive copies the data-name LONG-ITEM into &2. Since LONG-ITEM is not a period, the &STOW directive removes the word from the input. LONG-ITEM is output.

2. The &IF is true, causing the &NOTE to be executed.

3. The loop processes SHORT-ITEM and outputs it.

4. The &GET directive copies the period into &2. The subsequent &IF is then true, causing an exit from the macro before the &STOW.

CA-MetaCOBOL+ Input:

```
               SP     MOVE &1(S,L) TO :
                      MOVE &1 TO
                      &REPEAT
                         &GET &2                /* GET NEXT WORD
                      &UNTIL &2 = '.'           /* EXIT IF PERIOD
                         &STOW                  /* REMOVE WORD
                         &2                     /* OUTPUT WORD
                         &IF &2'S > 256         /* SIZE > 256?
                           &NOTE
                           &( &2 ' TOO LONG.' &)
                         &ENDIF
                      &ENDREP
                .  .  .
                DATA DIVISION.
                WORKING-STORAGE SECTION.
                77    SHORT-ITEM PIC X(256).
                77    LONG-ITEM PIC X(257).
                      .  .  .
                PROCEDURE DIVISION.
                .  .  .
                      MOVE SPACES TO LONG-ITEM
****NOTE  N99   LONG-ITEM TOO LONG.
                                 SHORT ITEM.
```

9.3 Analyzing Source Words

The Type attribute (&n'T) can simplify the task of parsing input in the macro model. It provides ready access to the type of word passed by the CA-MetaCOBOL+ Translator to the macro model. Reference to the Type attribute returns a one-character, alphanumeric value representing the syntax type of the current value of a symbolic operand. Type codes are:

Type	**Description**
&n'T = ` '	&n contains none of the following.
&n'T = `A'	&n contains an Area A indicator.
&n'T = `L'	&n contains a literal or figurative constant.
&n'T = `N'	&n contains a NOTE or comment.
&n'T = `S'	&n contains a String macro name.
&n'T = `V'	&n contains a verb or terminating period.

In order to be able to process a string of variable source words in a single macro model, follow these steps:

1. Copy to the next source word using an &GET.

2. Determine whether the word applies to the current macro using the Type attribute (or other attributes).

3. If applicable, remove the word from the source using &STORE or &STOW.

Embedded MOVE Example

In many cases, a parsing macro must interpret word strings which have no specific logical terminator, such as a period. Instead, the macro must look for the *beginning* word of some other function and terminate execution before that word is removed from the input. Such macro logic requires that the last &GET executed NOT be logically followed by an &STOW or &STORE, thereby retaining the word examined in the input.

The following is an improved version of the MOVE macro. Rather than test for terminating period, this version tests for the type of word copied and terminates the &GET/&STOW loop when a word type is encountered that is ineligible as a receiving field.

The macro uses one &REPEAT loop within another. The inner loop &GETs and passes through all COBOL notes and comments. The outer loop examines each source word copied by the &GET and terminates macro execution when the word is a verb or a period, a String macro name, an Area A indicator, a literal, or a $PDX prototype. ($PDX is a word used to invoke a macro at the end of the Procedure Division; see Section 11.1.) In this case, the word ELSE is a COBOL verb, and causes termination of the macro loop before the &STOW.

CA-MetaCOBOL+ Input:

```
          SP     MOVE &1(S,L) TO :
                 MOVE &1 TO
                 &REPEAT
                     &REPEAT
                         &GET &2               /* GET NEXT WORD
                     &UNTIL &2'T NE 'N'        /* SKIP NOTES
                         &STOW
                         &2
                     &ENDREP
                 &UNTIL &2'T NOT EQ ' ' &OR    /* EXIT IF VERB OR
.
                                               /* NEW MACRO, AREA
A,
                                               /* LITERAL, OR
                         &2 EQ '$PDX'          /* $PDX
                     &STOW
                     &2
                     &IF &2'S GT 256
                         &NOTE &( &2 ' TOO LONG' &)
                     &ENDIF
                 &ENDREP
           .  .  .
           DATA DIVISION.
           WORKING-STORAGE SECTION.
           77    SHORT-ITEM PIC X(256).
           77    LONG-ITEM PIC X(257).
           PROCEDURE DIVISION.
                 IF X = Y
                     MOVE SPACE TO
                                  LONG-ITEM
****NOTE      N99    LONG-ITEM TOO LONG.
                                  SHORT-ITEM
              ELSE
                 .  .  .
```

9.4 Parsing Control Words

There are several conditions which cause the &GET directive to copy control words which do not appear in the input source. It may be necessary, once such words are identified, to &STOW or &STORE them and substitute them in the output before processing the source words that follow. When such words are acquired and substituted by &GET and &STORE/&STOW, they affect the Translator as if they had been output directly from the input source or by prototype substitution.

Three types of words are described here:

- Area A indicators.
- Output control words.
- COBOL COPY and LIBRARIAN -INC text.

Area A Indicator

An Area A indicator is a control word that forces the next word to begin in Area A. It is not seen in the input listing. An &GET for a word appearing in Area A first copies an Area A indicator with Type attribute 'A'. After an &STORE or &STOW, a subsequent &GET is required to obtain the actual Area A word.

For example, given the following input source,

```
...
 GO TO READ-INPUT.
SUBROUTINE-X.
...
```

successive &GET and &STORE or &STOW directives acquire the current values and Type attributes as follows:

Current Value	**Type**
GO	V
TO	blank
READ-INPUT	blank
.	V
control-word	A (Area A indicator)
SUBROUTINE-X	blank
.	V
...	

Output Control Words

There are a number of instances when &GET acquires output control words that precede text - for example, accounting comments, debugging lines, line output, enabled comments, spacing controls, and out-of-line directives. All of these control words are Type 'N' notes.

To determine the precise classification of these control words, you can access &SETR register NOTE. The NOTE register reflects the status of the *most recent setting* by one of these word types. It, therefore, has no practical meaning unless the current word is determined to be a type note (&n'T='N').

For a description of &SETR, see Section 6.4.6. For a list of the word types returned in the NOTE register, see Appendix C.

Control Word Parsing Example

The next section describes how to place macro substitution words out-of-line from the input they replace. An &GET for a nested Word or Prefix macro which generates out-of-line code acquires an out-of-line control word with Type attribute 'N'. &SETR register NOTE returns the out-of-line directive executed.

CA-MetaCOBOL+ Input:

```
PP     SET= :
           MOVE '1' TO &
           &DATAWS            /* BEGIN OUT-OF-LINE
           77 & PIC X.
           &END
SP     IF  :  IF
           &REPEAT
               &GET &1
               .   .   .
           &ENDREP
           .   .   .
 PROCEDURE DIVISION.
 .   .   .
       IF COUNT EQUAL '0' SET=COUNT GO ...
       .   .   .
```

Successive &GET and &STOW or &STORE directives acquire the current values, Type attributes, and &SETR NOTE register values following:

Current Value	**Type**	**&SETR NOTE**
IF	V	
COUNT	blank	
EQUAL	blank	
'0'	L	
MOVE	V	
'1'	L	
TO	blank	
COUNT	blank	
control word	N	4(&DATAWS)
77	L	
COUNT	blank	
PIC	blank	
X	blank	
.	V	
control word	N	0(&END)
GO	V	
...		

COBOL COPY or LIBRARIAN-INC

You may wish to handle words from COPY books in a special manner. For example, you may want to output words from COPY books unchanged. How a COBOL COPY statement or LIBRARIAN-INC command is parsed by an &GET and &STOW/&STORE loop depends on the translate-time options specified.

- COBOL COPY

  If COPY=ACTIVE is specified, &GET acquires the copied text as though it were in the input stream and does not acquire the COPY statement. If the Translator cannot retrieve the library text for a COPY in the Data Division, a period is returned.

  If COPY=IGNORE or COPY=PASSIVE is specified, &GET acquires only the COPY statement and not the associated text.

- LIBRARIAN -INC

  If the -INC option is specified and if the LIBRARIAN -INC command is subordinate to a *&LIBED statement from the primary input file, the &GET acquires the included text as though it were from the input stream and does not acquire the -INC command.

  If the -INC option is not specified and/or if the -INC command is not subordinate to a *&LIBED statement from the primary input file, &GET acquires only the -INC command and not the associated text.

The input copied or included from external libraries can be identified by &SETR register STATUS. STATUS represents the source of the input.

```
0=Normal input.
1=COPY or -INC statement.
2=COPYed or -INCed text.
```

For example, a source library member named WKLIB contains:

```
01 WORK-AREA.
   02 WORK-KEY...
   ...
   02 WORK-END   PIC X.
```

Given the following input source,

```
01 WORK-RECORD   COPY WKLIB.
01 HOLD-AREA...
...
```

successive &GET and &STOW/&STORE directives will acquire the current values, Type attributes, and &SETR STATUS register values shown in the following tables.

Current Value	**Type**	**COPY=ACTIVE &SETR STATUS**	**Comment**
control-word	A	0	Area A Indicator
01	L	0	
WORK-RECORD	blank	0	
.	V	2	Period following WORK-AREA
02	L	2	
WORK-KEY	blank	2	
...	. . .	...	
02	L	2	
WORK-END	blank	2	
PIC	blank	2	
X	blank	2	
.	V	2	
control-word	A	0	Area A Indicator
01	L	0	
HOLD-AREA	blank	0	

	COPY=IGNORE or COPY=PASSIVE		
Current Value	**Type**	**&SETR STATUS**	**Comment**
control-word	A	0	Area A Indicator
01	L	0	
WORK-RECORD	blank	0	
COPY	blank	1	
WKLIB	blank	1	
.	V	1	Period following WKLIB
control-word	A	0	Area A Indicator
01	L	0	
HOLD-AREA	blank	0	

Alternatively, a CA-LIBRARIAN module named WKLIB contains:

```
01 WORK-AREA.
   02 WORK-KEY...
   ...
   02 WORK-END   PIC X.
```

Given the following input source,

```
-INC WKLIB
     01 HOLD-AREA...
     . . .
```

successive &GET and &STOW/&STORE directives will acquire the current values, Type attributes, and &SETR STATUS register values shown in the following tables.

	-INC option IS Specified		
Current Value	**type**	**&SETR STATUS**	**Comment**
control-word	A	2	Area A Indicator
01	L	2	
WORK-AREA	blank	2	
.	V	2	
02	L	2	
WORK-KEY	blank	2	
...	. . .	...	
02	L	2	
WORK-END	blank	2	
PIC	blank	2	
X	blank	2	
.	V	2	
control-word	A	0	Area A Indicator
01	L	0	
HOLD-AREA	blank	0	

-INC option NOT Specified

Current Value	**Type**	**&SETR STATUS**	**Comment**
-INC WKLIB	N	1	&SETR...NOTE=S(-INC)
control-word	A	0	Area A Indicator
01	L	0	
HOLD-AREA	blank	0	

Note: The preceding example is based upon the 1968 ANSI COBOL standard for COPY statements. The 1974 ANSI COBOL COPY requires slightly different source and/or library syntax. Further information concerning the COPY and -INC options can be found in the CA-MetaCOBOL+ *User Guide*.

9.5 Summary - Parsing Input Source

Listed below are the key points covered in this section.

- In order to handle the variety of ways input source can be formatted, models of String macros can control input parsing. Once a macro has been invoked by matching at least one source word, &GET can be used to copy the next word.
- &GET does not remove the word from the input stream; it merely saves it in a symbolic operand.
- Each word copied by &GET can be removed from the input stream by an &STORE or &STOW directive. &STORE removes only the next word. &STOW removes the next word and all of its qualifiers and subscripts.
- You must &STORE or &STOW before executing another &GET.
- Use Type attributes (&n'T) or other attributes to analyze the copied word.
- Use &GET and &STORE/&STOW in an &REPEAT loop to continuously copy, analyze, and remove source input.
- Control words inserted by CA-MetaCOBOL+ may be encountered by &GET.

  Control words can be identified using Type attributes and &SETR registers NOTE a status; they are then stored or stowed before subsequent words are processed.

Exercise 10: COBOL programmers must often define a Working-Storage area containing specific values, and then REDEFINE the area as a table. For example:

```
01 TABLE-AREA-1.
   02 FILLER     PIC X(14) VALUE 'ALABAMA'.
   02 FILLER     PIC X(14) VALUE 'ALASKA'.
   .  .  .
   02 FILLER     PIC X(14) 'WYOMING'.
01 TABLE-1       REDEFINES TABLE-AREA-1.
   02 STATE      OCCURS 50 TIMES
                 INDEXED BY STATE-INDEX
                 PIC X(14).
```

Write a macro that will create a table similar to this sample from the following input:

```
 TABLE 1 STATE STATE-INDEX X(14) 'ALABAMA`
'ALASKA' ...  'WYOMING'.
```

The macro prototype can be generalized as:

```
&TABLE level-number entry-name index-name
 picture-string literal-1 .  .  .  literal-n.
```

Three variables are required to:

1. Count the calls to the &TABLE macro.

2. Count the literals representing values. The count becomes the object of the OCCURS clause.

3. Compute the starting level-number plus 1.

Obtain and count values until a non-literal source word is encountered.

For exercise solutions, see Appendix A.

10. Model Programming - Controlling Output

For the most part, CA-MetaCOBOL+ automatically formats each line of output and places text in its proper location in the output stream. Standard COBOL formats, defined when CA-MetaCOBOL+ is installed or input as translate-time options, are observed automatically. There are times, however, when the programmer needs to control macro model output, either to override defaults or to supplement standard features. This section describes how macro models can control the format of the output line and the placement of text in the output stream.

CA-MetaCOBOL+ assumes that each output line follows standard COBOL Area A and Area B alignment rules. CA-MetaCOBOL+ options define other rules for indentation and line breaks following various COBOL words, such as division, section, and paragraph headers, COBOL verbs and clauses, continuation lines, and repetition and selection structures. The CA-MetaCOBOL+ *User Guide* describes these formatting controls completely.

This automatic line formatting can be overridden and extended by the macro model. Chapter 5 discussed directives to control Area A and Area B alignment. This chapter describes:

- Generating user-formatted output.

- Setting and using tabs in output lines (&SETTAB and &GOTAB).

- Setting margins in output lines (&MARGIN).

As a rule, CA-MetaCOBOL+ places output words in the same program location as the input words that they are replacing. The macro model can also override this rule. It may, for example, be desirable to build the output data descriptions while processing the input Procedure Division.

Macro model directives described here also allow this out-of-line placement of output words. The &NOTE directive, also introduced in Chapter 5, can be used to insert diagnostics in the output text. This section describes:

- Placing output in another COBOL division or section.

- Placing output (such as JCL and compiler directives) before or after the COBOL source or to another output file.

- Placing output in a macro-defined location.

10.1 Generating User-formatted Text

The CA-MetaCOBOL+ macro language contains special provisions for generating complete lines without reformatting. Such line output records can represent blank lines, COBOL comments, parameter records, assembly code, or dynamic macros to be input to a subsequent pass of the Translator.

Format:

```
L [text]
```

L
: must appear in column 7 as part of a macro model to generate a line output record.

text
: can be placed in columns 8-72 and becomes the content of the output record. Text is shifted left 7 characters so that column 8 in the line output statement becomes position 1 of the substituted record. For example:

CA-MetaCOBOL+ Input:

```
      L--------
```

CA-MetaCOBOL+ Output:

```
--------
```

Please note:

- Line output directed to COBOL output source has sequence numbers inserted as long as positions one through six are blank. COBOL output is truncated after column 72.

- All characters of line output in excess of 110 are truncated from the right. Line output records directed to 80-column output are truncated after column 80.

Since line output is not word oriented or COBOL-format oriented, constant characters and words have no meaning. As a result, unpaired parentheses, unpaired quotes, ampersands, directive expressions, etc., which would otherwise cause CA-MetaCOBOL+ to take action, are ignored in line output statements. Symbolic operands are not interpreted.

However, the names of alphanumeric and numeric variables are replaced in the line output record by current values. Columns are adjusted if the current value is longer or shorter than the variable name. The variable name need not be preceded or followed by a space, so that the rules for variable reference are non-standard, as follows:

- The line output text is examined from left to right. Whenever the characters &V are encountered, CA-MetaCOBOL+ assumes that a variable name follows.
- The character following &V is attached to &V as the variable name.
- Successive characters are attached to the variable name until a character less than "A" is encountered.

A variable name used in line output, therefore:

- Cannot be followed immediately by an alphabetic or numeric character (the longer name fails to match a variable name or matches the wrong variable).
- Cannot contain a special character in other than the first character following &V.

Comment Box Example

The $COM macro, below, builds boxed comments in columns 7-72. The macro also demonstrates techniques for character-filling or space-filling variables in order to create "fixed- format" line output. Note the treatment of the variable &VC to assure a space-filled, 60-character area for columns 12-71, forcing an asterisk in column 72.

CA-MetaCOBOL+ Input:

```
S     $COM &1(L) :
      &LOCAL &VH = NULL X(64)                 /* HYPHEN
      &LOCAL &VB = NULL X(64)                 /* BLANKS
      &LOCAL &VCOM X(64)                      /* COMMENT
      &IF &VH = NULL
         /*          1...5...10...15.
         &SET &VH =  '--------------'
         &SET &VB  = '              '
         &SET &VH = &( &VH &VH &)
         &SET &VH = &( &VH &VH &)
         &SET &VB = &( &VB &VB &)
         &SET &VB = &( &VB &VB &)
      ENDIF
      &SET &VCOM = &( '    ' &1 &VB &)
L        *&VH*
L        *&VB*
L        *&VCOM*
L        *&VB*
L        *&VH*
  .  .  .
 PROCEDURE DIVISION.
      &COM 'PROGRAM MAIN-LINE LOGIC.'
 MINA SECTION.
      .  .  .
```

CA-MetaCOBOL+ Output:

```
  .  .  .
 PROCEDURE DIVISION.
*------------------------------------------*
*                                          *
*      PROGRAM MAIN-LINE LOGIC.   . . .    *
*                                          *
*------------------------------------------*
 MAIN SECTION.
      .  .  .
```

10.2 Reformatting Output

CA-MetaCOBOL+ uses new lines and indentation to reflect the logic of the output source. Since individual applications may have unique structures, individual macro models may require their own output formatting. The macro model can use the directives described here to perform functions such as aligning a series of file-names or data-names, indenting IF - ELSE - ENDIF structures, or otherwise highlighting the logic of a program.

These directives define tab registers and align individual words by column position. They can redefine the left margin, so that all continuation lines thereafter are aligned by column position. Tabs and margins can be set either at absolute column locations or at locations relative to the current indentation level.

Aligning by Tab Registers

To align output words at specific columns, use horizontal tab registers. The &SETTAB directive defines a tab register. The &GOTAB directive forces the next word to begin at a previously defined tab register.

Format:

```
&SETTAB register = column

&GOTAB register
```

&SETTAB

defines a tab register at column position column.

Absolute tab registers A0 through A15 can be equated to an integer or variable representing a column position 8 through 72. Relative tab registers R0 through R15 can be equated to a number of column positions to the right of the current left margin.

&GOTAB

applies only to the word following it.

Please note:

- The default values of all tab registers are 0, causing no change in automatic formatting rules.
- If the &GOTAB cannot be satisfied on the current line, the &GOTAB is attempted on the next line. The word may be left-shifted if it would cross into column 73.
- A tab register set at less than absolute column 8 causes &GOTAB to be ignored. A value for the column of more than 72 causes the next word to be right-justified on a new line.

The &SETTAB directive can be "generated" out-of-line to control formatting of code based upon the longest name in a series, the MOVE statement with the longest sending operand, etc.

Tab Alignment Example

Assume that all CALL operands are to be aligned vertically under the first, relative to the indentation of the CALL verb.

The following macro sets a relative tab register R0 at twelve positions plus the length of the subroutine name past the current left margin. The &REPEAT loop acquires each operand, skips a line using &B, tabs to R0, and outputs the operand.

CA-MetaCOBOL+ Input:

```
SP      CALL &1 USING &2(S) :
        CALL &1 USING &2
        &LOCAL &V-INDENT 9(2)
        &SET &V-INDENT = 12 + &1'N
        &SETTAB R0 = &V-INDENT
        &REPEAT
        &GET &2
&UNTIL &2'T NE ' '
             &STOW
             &B
             &GOTAB R0
             &2
        &ENDREP
 .  .  .
 &PD
        CALL SUBRTN USING A B C.
        .  .  .
```

CA-MetaCOBOL+ Output:

```
 .  .  .
 PROCEDURE DIVISION.
        CALL SUBRTN USING A
                          B
                          C.

        .  .  .
```

Setting the Left Margin

Tab stops are effective for positioning one word at a time. To align a series of words or a series of continuation lines under one column position, reset the left margin using the &MARGIN directive.

Format:

```
&MARGIN register
```

&MARGIN

applies only to the continued lines following it. It is cancelled by any automatic format control.

The register can be A0 through A15 for an absolute margin, or R0 through R15 for a relative margin.

Please note:

- A relative margin is computed relative to the current left margin established by the automatic formatting.

- The default values of all tab registers are 0. A value of 0 in the referenced tab register will cause &MARGIN to be ignored.

- A value for the margin of less than 8 will cause &MARGIN to be ignored. A value for the margin of more than 72 will cause the next word to be right justified on a new line.

Margin Alignment Example

Assume that all call operands are to be aligned vertically under the first, relative to the indentation of the CALL verb. The following macro uses one &MARGIN in place of the previous example's repeated &GOTABs.

CA-MetaCOBOL+ Input:

```
SP      CALL &1 USING &2(S) :
        CALL &1 USING &2
        &LOCAL &V-INDENT 9(2)
        &SET &V-INDENT = 12 + &1'N
        &SETTAB R0 = &V-INDENT
        &MARGIN R0
        &REPEAT
            &GET &2
        &UNTIL &2'T NE ' '
            &STOW
            &B
            &
        &ENDREP
 .  .  .
 $PD
        CALL SUBRTN USING A B C.
        .  .  .
```

CA-MetaCOBOL+ Output:

```
        .  .  .
        PROCEDURE DIVISION.
            CALL SUBRTN USING A
                              B
                              C.
            .  .  .
```

10.3 Out-of-Line Substitution

Out-of-line directives place all substitution words that follow at specified locations in the output COBOL source program rather than directly in place of the source statement that calls the macro.

When writing COBOL programs, the programmer often finds it desirable to place statements out-of-line. For example, while coding the Procedure Division, additional Working-Storage entries may be required. CA-MetaCOBOL+ macros can be defined to automatically handle required work areas, subroutines, etc., so that the programmer need not be concerned with their specification. In addition, it may be desirable to generate JCL or comments before or after the COBOL source from within the macro model.

Once an out-of-line directive is executed, all substitution words and line output records appear in the specified location in the order in which they are generated until out-of-line substitution is turned off.

The CA-MetaCOBOL+ out-of-line directives and their associated predefined locations are shown in the table on the following page. The rest of this section describes:

- Ending out-of-line substitution.
- Continuing out-of-line substitution.
- Generating output to standard COBOL locations, non-COBOL locations, and macro defined locations.

Directive	Description
&ANTE	Subsequent output is placed before the first generated COBOL Division header. (&ANTE should be used only in conjunction with Line output.)
&AUX	Subsequent output is directed to the Auxiliary Output File, as well as the Listing file. (&AUX should be used only in conjunction with Line output.)
&AUXN	Subsequent output is directed to the Auxiliary Output File with the AUXILIARY OUTPUT listing suppressed.
&DATA	Subsequent output is directed to the end of the Data Division following all other Data Division out-of-line code.
&DATAF	Subsequent output is directed to the end of the File Section.
&DATAL	Subsequent output is directed to the end of the Linkage Section.
&DATAR	Subsequent output is directed to the end of the Report Section.
&DATAW	Subsequent output is directed to the end of the Working-Storage Section.
&DATAWS	Subsequent output is directed to the beginning of the Working-Storage Section.
&DATAWX	Subsequent output is directed to the end of the Working-Storage Section after all &DATAW output.
&ENV	Subsequent output is directed to the end of the File-Control paragraph or, in its absence, to the end of the Environment Division.
&POST	Subsequent output is directed to the end of the generated output, after any &PROC, &PROCS, and &PROCX output. (&POST should be used only in conjunction with Line output.)
&PROC	Subsequent output is directed to the end of the Procedure Division, after any &PROCS output.
&PROCS	Subsequent output is directed to the end of the current Procedure Division section or to the end of the Procedure Division if there are no sections.
&PROCX	Subsequent output is directed to the end of the Procedure Division, after any &PROCS and &PROC output and before any &POST output.

Ending Out-Of-Line Substitution

Out-of-line substitution is explicitly terminated by:

- Another out-of-line directive.
- The &END directive.
- The end of the current macro (unless &NOEND is active).
- An error which terminates model execution.
- The end of a macro that has issued an out-of-line directive. For example:

A String macro model substitutes a word that calls a nested Word macro. The Word macro issues an out-of-line directive. When the nested Word macro ends, the mode is reset to in-line, and control returns to the String model.

If out-of-line mode is set by the String macro, the nested Word macro is directed out-of-line. Out-of-line mode remains in effect when control returns to the String macro mode. (However, if the Word macro also issues an out-of-line directive, the mode is reset to in-line when control returns to the String macro).

The &END directive turns off out-of-line generation.

Format:

```
&END
```

&END
resets the out-of-line mode to in-line generation. If already in the in-line mode, executing &END has no effect and processing continues.

Continuing Out-Of-Line Substitution

The &NOEND out-of-line directive continues out-of-line mode after termination of a String macro.

Format:

```
&NOEND
```

&NOEND
The only way to remain in out-of-line mode when terminating execution of a String macro model is to execute the &NOEND directive after an out-of-line directive.

This directive is valid ONLY in a String macro model. Once executed, &NOEND cancels the automatic resetting of the out-of-line mode to in-line at the end of a String macro model until it is explicitly turned off by another out-of-line directive.

Please note:

- &NOEND affects the source of input as well as the destination of output. Macros that follow the &NOEND must have the same division code as the division specified in the still-active out-of-line directive. CA-MetaCOBOL+ will not execute macros from any other divisions until &NOEND is turned off.

Outputting to Standard COBOL Locations

All out-of-line directives except &ANTE, &POST, &AUX, and &AUXN direct subsequent model substitution words to pre-defined points within the generated COBOL program. Use them to:

- Generate complete file specifications (SELECT sentences and FD entries) from a single source statement.
- Define work areas as required within Procedure Division logic.
- Define complex verb-like macros which can be executed conditionally by substituting a PERFORM statement in-line and the subroutine at the end of the current section or Procedure Division.

To simplify such out-of-line substitution within the COBOL source, all Word and Prefix macro calls appearing in words substituted out-of-line are expanded as though they were found in the source for the appropriate division.

Generate SELECT and FD Example

As an example of out-of-line substitution, a statement coded as,

```
        URFD file-name SYSnnn device record-name
```

can be placed in the Data Division, File Section, to generate the proper SELECT entries in the Environment Division and FD entries in the Data Division for a VSE UNIT-RECORD file.

CA-MetaCOBOL+ Input:

```
          SD     URFD &1 &2 &3 &4 :
                 &ENV
                 SELECT &1 ASSIGN TO &( &2 -UR- &3 -S &).
                 &END
                 FD &1
                      RECORDING MODE F
                      LABEL RECORD OMITTED
                      DATA RECORD &4.
                 01 &4
           .  .  .
           ENVIRONMENT DIVISION.
           .  .  .
           FILE CONTROL.
           .  .  .
           DATA DIVISION.
           FILE SECTION.
                 URFD CARD-READER
                                 SYS006 254OR READER-RECORD PIC
X(80).
                 URFD PRINTER
                                 SYS008 1403 PRINTER-RECORD.
                 02 FILLER PIC X.
                 02 PRINTER-IMAGE PIC X(132).
           .  .  .
```

CA-MetaCOBOL+ Output:

```
           .  .  .
           ENVIRONMENT DIVISION.
           .  .  .
           FILE-CONTROL.
                 SELECT CARD-READER      ASSIGN TO SYS006-UR-254OR-S.
                 SELECT PRINTER          ASSIGN TO SYS008-UR-1403-S.
           .  .  .
           DATA DIVISION.
           FILE SECTION.
           FD CARD-READER                RECORDING MODE F
                                         LABEL RECORD OMITTED
                                         DATA RECORD READER-RECORD.
           01 READER-RECORD              PIC X(80).
           FD PRINTER                    RECORDING MODE F
                                         LABEL RECORD OMITTED
                                         DATA RECORD PRINTER-RECORD.
           01 PRINTER-RECORD
                 02 FILLER               PIC X.
                 02 PRINTER-IMAGE        PIC X(132).
           .  .  .
```

Out-of-Line to the Data Division

A number of out-of-line directives generate data items to locations within the Data Division. These items can be made available to data item acquisition functions (e.g., &SCAN) and data attribute analysis.

Use the &DSTART directive to tell the Translator that, though translation is currently in another division, the items that follow are to be treated as data items. Data attribute building for data item definitions directed out-of-line is terminated by &DSTOP.

Format:

```
&DSTART

&DSTOP
```

&DSTART and **&DSTOP**
: are not, in themselves, out-of-line directives. They define the start and end of data attribute building for items directed out-of-line to the Data Division.

Relocating Data Items from the Procedure Division Example

Procedure Division code between START DATA and END DATA can be relocated to the end of Working-Storage by the following macros.

CA-MetaCOBOL+ Input:

```
SP     START DATA :
       &DSTART
       &DATAW
       &NOEND
SD     END DATA :
       &END
       &DSTOP
 .  .  .
 $PD
 MAIN-LINE-LOGIC.
       START DATA.
 01 END-OF-FILE PIC X.
       END-DATA.
 .  .  .
```

Note that using &NOEND after &DATAW forces the macro writer to assign the END DATA macro to the Data Division.

Outputting to Non-COBOL Locations

The out-of-line directives &ANTE, &POST, &AUX, and &AUXN direct subsequent model substitution words ahead of the generated COBOL program, behind the generated COBOL program, and to the Auxiliary File, respectively. (The Auxiliary File is an extra file available to the programmer.) Use them to:

- Generate Job Control Language statements.
- Create parameter records.
- Produce assembly language modules.
- Build supplemental translation macros for a second execution of CA-MetaCOBOL+.
- Report translation results to the Auxiliary Listing for programmer review.

Line output is used with these out-of-line directives. Since this is not COBOL being generated, the Translator will produce records of unpredictable form.

Line output records directed out-of-line by &ANTE, &POST, and &AUX and containing either ** or */ in positions 1 and 2 are printed on the appropriate listing, but suppressed from the corresponding output file. In addition, */ in positions 1 and 2 forces a page eject on the appropriate listing before printing. These facilities are included to generate summary reports.

Auxiliary Report Example

The following standards-auditing macros display violations, such as use of the ALTER verb, on a separate report. Diagnostics must include the sequence number of the source record containing the violations, obtained from the special register SEQ using the &SETR directive. The report must contain a heading line, including the program-name.

CA-MetaCOBOL+ Input:

```
SI     PROGRAM-ID.  &1.  :
       &LOCAL &VID X(8)
       &SET &VID = &1
       &AUX                     /* OUT TO AUX FILE
L*/    PROGRAM &VID DIAGNOSTIC LIST
L**
       &END
       PROGRAM-ID.  &1.
WP     ALTER :
       ALTER
       &LOCAL &VLINE X(6)
       &SETR &VLINE = SEQ       /* GET SEQUENCE NO.
       &AUX                     /* OUT TO AUX FILE
L**    ALTER VERB IN STATEMENT &VLINE
       &END
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID.  PREDIT.
.  .  .
000400 PROCEDURE DIVISION.
.  .  .
000500 ALTER PARA-A TO ...
.  .  .
```

Auxiliary Listing:

```
*/ PROGRAM PREDIT DIAGNOSTIC LIST
**
** ALTER VERB IN STATEMENT 000500
.  .  .
```

Outputting to Macro-Defined Locations

The out-of-line directives explained thus far direct substitution words to certain pre-established points in the output. A second level of directives lets you define out-of-line locations and direct substitution words to them. Use the directive expressions &MARKER and &POINT.

Format:

```
&MARKER {&Vname }
        {literal}
```

literal or &Vname

The current value of literal or &Vname can range from 0 through 4999, permitting an additional 5000 out-of-line locations. The &MARKER is inserted at the potential locations for an out-of-line substitution.

&MARKER
is not an out-of-line directive, since it merely "marks" a potential location for out-of-line insertion.

&POINT

The &POINT out-of-line directive expression directs subsequent substitution words to the output location defined by an &MARKER of the same number. The required format is:

```
&POINT {&Vname }
       {literal}
```

literal or &Vname

The current value of literal or &Vname can range from 0 through 4999 and must correspond to an &MARKER of the same value. Please note:

- As with other out-of-line directives, &END terminates output to the &POINT location.
- More than one location can have the same &MARKER. In such a case, substitution words following a matching &POINT are generated in the output at ALL locations defined by such &MARKERs.
- Similarly, more than one &POINT can direct output to a single &MARKER location. In such a case, output from a later pointer is directed immediately BEHIND the previous pointer's output.
- Statements generated out-of-line to another COBOL division via the &POINT directive expression may not be formatted correctly, since certain formatting logic within the Translator assumes the current division. Proper format can be achieved by preceding the &POINT directive with an explicit, division-oriented out-of-line directive, such as &ENV, &DATA, or &PROC.

CLOSEALL Example

The CLOSEALL function was presented earlier as an exercise to demonstrate global variable usage and conditional logic. In it, the CLOSEALL word is replaced by a COBOL CLOSE statement for all defined files. The same function can be accomplished using &MARKER and &POINT directives instead of global variables.

The SELECT macro outputs the SELECT clause in-line, then uses &POINT to output the data-name in &1 to location 1. Location 1 is defined in Word macro CLOSEALL by an &MARKER following output of the COBOL CLOSE verb.

Note that using &POINT and &MARKER is normally confined to very complex transactions, where other CA-MetaCOBOL+ facilities cannot be used to achieve the proper results. Less complex situations can usually use other techniques.

CA-MetaCOBOL+ Input:

```
SE     SELECT &1 :
       SELECT &1
       &POINT 1
       &1
       &END
WP     CLOSEALL :
       CLOSE
       &MARKER 1
 .  .  .
 ENVIRONMENT DIVISION.
 .  .  .
 FILE-CONTROL.
       SELECT FILE-A...
       SELECT FILE-B...
       SELECT FILE-C...
       SELECT FILE-D...
 .  .  .
 PROCEDURE DIVISION.
 .  .  .
       CLOSEALL...
 .  .  .
```

CA-MetaCOBOL+ Output:

```
 ENVIRONMENT DIVISION.
 .  .  .
 FILE-CONTROL.
       SELECT FILE-A...
       SELECT FILE-B...
       SELECT FILE-C...
       SELECT FILE-D...
 .  .  .
 PROCEDURE DIVISION.
 .  .  .
       CLOSE FILE-A FILE-B FILE-C FILE-D ...
 .  .  .
```

Whenever a choice between out-of-line substitution and some other technique exists to perform the identical function, the macro writer should recognize that extensive use of out-of-line facilities increases translation time of CA-MetaCOBOL+.

10.4 Summary - Controlling Output

Listed below are the key points covered in this section.

- The macro model can control the format of output lines and the placement of output text.
- User-formatted line output can be used to produce blank lines, comments, assembly code, etc. that is not to be changed to COBOL format. Such line output begins with an L in column 7.
- The format of output source can be controlled by these directives:

`&SETTAB and &GOTAB`	- align next word by tab stop.
`&MARGIN`	- aligns continuation lines by left margin.

- Out-of-line substitution directs output to a location other than that of the words being replaced. Output can be directed to:
  Standard COBOL locations.

  Locations before or after COBOL source.

  The Auxiliary file.

  Specified locations defined by &MARKER and accessed by &POINT.
- &END explicitly terminates out-of-line substitution.

  Normally, the end of a macro also terminates out-of-line substitution. &NOEND continues out-of-line substitution at the end of a String macro.

- &DSTART and &DSTOP allow you to retrieve data-names and attributes of data items substituted out-of-line to the Data Division.

Exercise 11: Write a String macro which can be executed during translation of any COBOL division to relocate any statement coded as,

```
$77 data-name PICTURE picture VALUE value.
```

to the beginning of the Working-Storage Section as the level-77 data item,

```
77 data-name PICTURE picture VALUE value.
```

For exercise solutions, see Appendix A

11. Predefined Prototypes - Using Event-Dependent Macros

The previous sections describe macro model programming. This section returns to the macro prototype. Macro prototypes have been presented as precise descriptions of words and word sequences to be treated as search arguments. There are situations, however, when a macro must gain model control when certain *events* occur during translation of the source, rather than when specific words and word patterns are detected.

CA-MetaCOBOL+ contains provisions for event-dependent macro translation via special String macro prototypes. Such prototypes, consisting of single-word macro names, are considered to be reserved macro names by the CA-MetaCOBOL+ Translator and must be used only for the specific purposes defined here.

The function of event-dependent prototypes is to initiate execution of the associated model when one of the following events is encountered:

- End of the input source program ($PD and $PDE).
- End of the Data Division ($DDX and $DDE).
- Data Division level number ($-LEVEL).
- Procedure-name definition ($-PROC).
- Any word, within the appropriate division, defined as a verb ($-VERB).

Event-dependent macro prototypes need not match actual source code, but are expanded whenever the condition each represents exists in the input being translated.

Note: Again, these prototypes are available in String macros only.

11.1 End of Input Source

The following reserved prototypes cause execution of the associated model at the end of the input source program:

Format:

```
S [P] {$PDE}
      {$PDX}
```

The prototype must be part of a String macro with a *P* division code. The precise event which calls the *$PDE* or *$PDX* macro is the detection of the end of input during Translation.

A maximum of 9 $PDX macros can be specified during a single translation. They are executed in the order of input. Only the last $PDE macro loaded is executed. If both are used, $PDE is called before any $PDXs.

When a $PDX macro is used, other String macros can also detect the end of input by using an $GET directive expression to look for the literal '$PDX'. The $PDX and $PDE macros are useful for summarizing translation results, often by interrogating many global variables and performing complex reporting to the Auxiliary listing.

Missing STOP RUN Example

The following macros issue a diagnostic if a STOP RUN statement is not found in the Procedure Division.

CA-MetaCOBOL+ Input Listing:

```
          SP     STOP RUN :
                 STOP RUN
                 &GLOBAL &VSR X             /* STOP RUN SWITCH
                 &SET &VSR = 'X' SP
                 $PDX :                     /* CALL AT END OF
INPUT
                 &IF &VSR NOT = 'X'
                     &NOTE &( 'PROGRAM CONTAINS NO STOP RUN' &)
                 &ENDIF
           .  .  .
           PROCEDURE DIVISION.
           .  .  .
****NOTE N99 PROGRAM CONTAINS NO STOP RUN
```

11.2 Level Number

Data Division level numbers take the form of numeric literals. As such, they cannot be defined as macro names. It is often necessary, however, to modify the Data Division item descriptions via CA-MetaCOBOL+.

The event that results in a call to the $-LEVEL macro is finding a numeric literal preceded by a period, or a period and an Area A indicator, in the Procedure Division.

Format:

```
SD     $-LEVEL  :
```

The prototype must be part of a String macro with a D division code.

Please note:

- Following a $-LEVEL macro calls, the level number itself must be obtained via an &GET directive expression.
- If the $-LEVEL macro is used to process an entire data definition, the terminating period must NOT be removed by an &STOW or &STORE directive. Since the $-LEVEL event is defined as a numeric literal preceded by a period, removal of the preceding period results in no call to the &-LEVEL macro for the subsequent level number.

Correcting REDEFINES Example
Consider converting redefined data items from IBM Levels D, E, and F COBOL to American National Standard (ANSI) COBOL. Under the older compilers, the object of the REDEFINES clause is ignored - i.e., no matter what name is specified as the area to be redefined, the compiler simply refers to the previous item of the same level number. ANSI COBOL, however, requires that the object of the REDEFINES clause contain the name of the previous item of the same level which does not contain a REDEFINES clause. All compilers require that the REDEFINES clause, if specified, must appear before other attribute clauses (i.e., as the third word of the item description).

The $-LEVEL macro can be used to accomplish this conversion function easily.

CA-MetaCOBOL+ Input:

```
SD     $-LEVEL :                    /* RUN FOR LEVEL NO.
       &LOCAL &VBASE(49) X(30)      /* DATA-NAME TABLE
       &LOCAL &VX       99          /* LEVEL INDEX
       &GET &1                      /* GET LEVEL NUMBER
       &IF &1 <50                   /* EXIT IF NOT 01-49
          &STORE
          &1
          &SET &VX = &1
          &GET &1                   /* GET DATA-NAME
          &STORE
          &1
          &GET &2                   /* GET NEXT WORD
          &IF &2 = 'REDEFINES`      /* IF REDEFINES,
             &STORE                 /*  THEN REMOVE
             &2                     /*  & OUTPUT
             &GET &2                /*  GET OBJECT
             &STORE
             &VBASE(&VX)
          &ELSE                     /* ELSE
             &SET &VBASE(&VX) = &1  /*  SAVE DATA-NAME
          &ENDIF
       &ENDIF
 .  .  .
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01    WORK-AREA.
       02 AREA1.
       02 AREA2 REDEFINES AREA1.
       02 AREA3 REDEFINES AREA2.
       .  .  .
```

CA-MetaCOBOL+ Output:

```
.  .  .
DATA DIVISION
WORKING-STORAGE SECTION
01     WORK-AREA.
       02 AREA1.
       02 AREA2 REDEFINES AREA1.
       02 AREA3 REDEFINES AREA1.
       .  .  .
```

11.3 Other Event-Dependent Macros

Other event-dependent macros include the following:

```
SD {$DDE}
   {$DDX}
```

This prototype must be part of a String macro with a *D* division code. This macro gets control at the end of the Data Division. It is actually triggered when the Procedure Division header is detected, but before it is processed.

A maximum of 9 $DDX macros can be specified during a single translation. They are executed in the order of input. Only the last $DDE macro loaded is executed. If both are used, $DDE is called before any $DDXs.

&DDX can be used to finish processing the Data Division, for example, to create section headers, to generate data definitions, and to guarantee in-line placement of Data Division entries.

```
SP    $-PROC  :
```

This prototype must be part of a String macro with a *P* division code. This macro is called every time a word is found in Area A in the Procedure Division.

$-PROC can be used to insert statements at the beginning of paragraphs, to count paragraphs, to place EJECT commands ahead of sections, and to audit the Procedure Division for proper Area A usage.

```
S[division-code]    $-VERB  :
```

This prototype must be part of a String macro with the *division-code* specified. During translation, it gets control every time a word having a Verb Type attribute (&N'T='V') or a terminating period is recognized.

For more information on the use of the event-dependent macros, refer to the *Macro Facility Reference.*

11.4 Summary - Using Event-Dependent Macros

Listed below are the key points covered in this section.

- Event-dependent macros are reserved, String-macro prototypes that gain control when specific events occur.
- $PDX and $PDE gain control at the end of the input source program.
- $DDX and $DDE gain control at the end of the Data Division.
- $-LEVEL gains control when a Data Division level number is encountered.
- $-PROC gains control when an Area A word is encountered in the Procedure Division.
- $-VERB gains control when a verb is encountered in any division.

Exercise 12: Today's compilers are more tolerant than in the past and this sometimes results in misplaced statements in the Procedure Division. Write a macro that will move verbs coded in Area A back to Area B.

12. Macro Writing Techniques - Standards and Debugging

12.1 Macro Writing Standards

The CA-MetaCOBOL+ macro facility possesses the physical attributes of a programming language. As with any other programming technique, applications development, debugging, and maintenance can be simplified by observing a few simple programming standards.

This section describes suggested standards for macros and macro sets. It includes:

- Macro set organization.
- Macro format.
- Macro naming conventions.

Macro Set Organization

The standard at Computer Associates, which may prove useful for other installations, is to divide the macro set into three basic components: the prologue, external and global variable definitions, and translation macros.

Prologue

Each macro set begins with commentary known as the *prologue*, which describes such subjects as the purpose of the set, the restrictions on use with other sets, and an explanation of diagnostics. The following figure is a skeletal sample of a prologue.

```
*$HDR name           title                     -version
*--------------------------------------------------------------*
*                                                              *
*  PURPOSE:                                                    *
*     Description of translation purpose.                      *
*                                                              *
*  DATE-WRITTEN:                                               *
*     Date.                                                    *
*                                                              *
*  VERSION:                                                    *
*     CA-MetaCOBOL+ Version.Release.Modification level.        *
*                                                              *
*  HISTORY:                                                    *
*     Justification for modification levels.                   *
*                                                              *
*  AUTHOR:                                                     *
*     Author name, address, telephone.                         *
*                                                              *
*  TRANSLATE-TIME OPTIONS:                                     *
*    Required/recommended translate-time options               *
*                                                              *
*  RESTRICTIONS:                                               *
*     All restrictions on use of the macro set                 *
*                                                              *
*  DIAGNOSTICS:                                                *
*     Text of each message, cause, and corrective actions;     *
*      or where to find documentation of diagnostics.          *
*                                                              *
*--------------------------------------------------------------*
```

A description of each item defined in the prologue follows:

**$HDR.* The distributed name of the macro set, the title of the macro set, and the version/release number of the latest update.

PURPOSE. A brief description of the type of translation performed. For example:

```
PURPOSE:
TO AUDIT STANDARD COBOL PROGRAMS FOR VIOLATIONS
OF
PROGRAMMING STANDARDS AND DIAGNOSE WHERE FOUND.
```

DATE-WRITTEN. As in the COBOL DATE-WRITTEN paragraph.

VERSION. CA-MetaCOBOL+ Translator Version and release required to perform the translation, as well as the current modification level of the macro set. For Example:

```
VERSION:
  V1.0.MO
```

The above specifies that CA-MetaCOBOL+ Version 1, Release 0 or later is required. No modifications have been made since the macro set was written.

HISTORY. An explanation of each modification, including the date applied. For example:

```
HISTORY:
   M1.SUMMARY REPORT ADDED.  09/15/91.
```

AUTHOR. Author name, address, installation, etc., if applicable.

TRANSLATE-TIME OPTIONS. A list of the CA-MetaCOBOL+ translate-time options required or recommended, or a reference to the appropriate documentation. For example:

```
TRANSLATE-TIME OPTIONS.
    COPY=ACTIVE REQUIRED.
        OTHERS OPTIONAL.
```

RESTRICTIONS. A list of the restrictions upon use of the macro set. Macro, variable, and label names which may conflict with other macro sets should be indicated. Compiler dependency, if any, should be listed. In addition, the rules for generated name formation should be specified. For example:

```
RESTRICTIONS:
   $-LEVEL, $-PROC MACROS.
   ALL OTHER NAMED MACROS.
   VARIABLE/LABEL NAMES BEGINNING &VA@/&L@.
   GENERATED NAMES BEGINNING ZZ-ABC-.
```

If there are no restrictions, this item may be omitted. If restrictions are noted in formal user documentation, the document should be referenced.

DIAGNOSTICS. Diagnostics not documented should be listed with an explanation of meaning and appropriate corrective action for example:

```
DIAGNOSTICS:
   ...
   STD08W-SHORT PROCEDURE-NAME.
      PROCEDURE-NAMES OF LESS THAN 6 ALPHANUMERIC
      CHARACTERS ARE OFTEN NOT DESCRIPTIVE OF
PROCESSING
      LOGIC.  EXPAND TO MORE MEANINGFUL NAME
UNLESS
      JUSTIFIABLE.
```

If diagnostics are documented elsewhere, this item should reference the appropriate documentation.

External and Global Variable Definitions

It is recommend that your macro sets defined all external and global variables in dummy Word macros (named, for example, $$EXTERN and $$GLOBAL, respectively). These macros follow the prologue and precede all translation macros, so that external and global variables may be located easily during macro set maintenance. Variable definitions are the only model statements within the $$EXTERN and $$GLOBAL macros, so that they do not affect translation if they are executed inadvertently.

Macro Organization

Macros should be organized in the sequence of standard COBOL for ease of reference. Within each macro set, the order is:

1. Word macros which apply to all divisions in alphabetical order.
2. Identification Division macros in the order of specification in a COBOL program.
3. Environment Division macros in the order of specification in a COBOL program.
4. Data Division macros in the order of specification in a COBOL program.
5. The $DDX macros, if specified.
6. Procedure Division macros in alphabetical order or the order of required specification in a CA-MetaCOBOL+ source program, as applicable.
7. The $PDX macros, if specified.

Macro Format

Unlike COBOL, macro models can be written in relatively free format. Standards for macro model format, therefore, are critical for macro debugging and maintenance. Three basic standards are recommended:

- Use only one statement per macro model line.
- Align logical destination definitions so that they are easily located.
- Begin directive expressions and substitution words in a standard column.

As an example, the model coding for the following macros is identical. Notice that the format standards add necessary clarity to the second macro definition:

Non-standard Format:

```
SE     SELECT &1 : SELECT &1 &GLOBAL &VFILE(9)X(30) &LOCAL &VX 99
       &SET &VX=&VX + 1 &IF &VX > 9 &NOTE &( 'FILE NAME ' &1 'NOT
       INCLUDED' &) &ELSE &SET &VFILE(7VX) = &1 &ENDIF
```

Standard Format:

```
       SE    SELECT &1 : SELECT &1
                   &GLOBAL &VFILE(9) X(30)
                   &LOCAL &VX 99
                   &SET &VX = &VX + 1
                   &IF &VX > 9
                       &NOTE &( 'FILE NAME ' &1 ' NOT INCLUDED' &)
                   &ELSE
                        &SET &VFILE(&VX) = &1
                   &ENDIF
```

Naming Conventions

Standards for diagnostic formation and generated COBOL names are important for macro set maintenance and standardization of generated COBOL code.

Diagnostic Message Notation

Begin &NOTE diagnostics with a prefix identifying the macro set, diagnostic number, and severity. For example:

STD08W-SHORT PROCEDURE-NAME

The first characters, *STD*, are the macro set abbreviation.

08 represents the diagnostic number, which can be used to isolate the diagnostic in the macro set prologue. Diagnostic numbers are assigned sequentially starting with 01.

The character *W* indicates a warning. Severity indicators might include:

Code	**Severity**
E	ERROR
W	WARNING
A	ADVISORY

You can also define condition codes to be passed from the Translator. Refer to the &COND directive expression in the *Macro Facility Reference*.

Generated Names

Where possible, the simplest and most satisfactory approach to COBOL name formation is to use a standard prefix for data-names and suffix for procedure-names. This convention requires the COBOL programmer to remember only the prefix or suffix and allows names to be easily changed by a simple Prefix macro. Examples are:

```
MM-data-name
```

and

```
procedure-name-EXIT
```

Data-names prefixed by MM could be easily expanded by a Prefix macro, such as the following:

```
P    MM-& : &( MASTER- & &)
```

12.2 Macro Debugging

The testing and debugging of CA-MetaCOBOL+ macros and macro sets is very similar to the testing and debugging of problem programs written in any language. Macro set debugging parallels integrated systems testing, where individual macros or programs must interrelate within the entire macro set or system.

This subsection discusses many of the techniques and facilities for macro debugging, including:

- Interpreting test results.
- Developing special-purpose debugging aids.
- Using the CA-MetaCOBOL+ macro debugging aid feature.

Normal programming disciplines which apply to macro writing, such as careful definition of the problem to be solved, flowcharting and desk checking, are beyond the scope of this chapter. Also not covered is interpretation of CA-MetaCOBOL+ diagnostics, since a thorough description of the cause and corrective action for each diagnostic is detailed in the CA-MetaCOBOL+ *User Guide.*

As an aid to Macro debugging, in addition to the output source listing, CA-MetaCOBOL+ provides a listing of macro definitions and the input source program. This Input and Diagnostics listing includes a STATISTICS listing at the end indicating the presence or absence of Translator errors and warnings. All errors should be corrected, if possible, before analyzing test results further.

Following correction of errors and warnings, CA-MetaCOBOL+ input and output must be examined to ensure correct translation. Many common errors can be corrected by a thorough analysis of the Input and Diagnostics listing. Others may require additional testing under control of user-defined or CA-MetaCOBOL+ debugging aids.

Trouble Shooting Guide

The following paragraphs list suggestions for analyzing many common logical errors.

No Apparent Call to a Macro

Failure to match source to a prototype is generally caused by:

- Failure to load the macro.
- P, S, or W type code not in column 7.
- Improper division codes in columns 8-11.
- Type code inconsistent with macro prototype.
- Improper spelling of prototype keywords or source words.
- Incorrect symbolic operand qualification in a String macro prototype.
- Specification of keywords OF, IN, or parentheses after a qualified symbolic operand in a String macro prototype.
- Identical macro prototype defined elsewhere.
- A Prefix macro alters a Word or String macro name or a String macro keyword.
- A Word macro overrides a String macro name or alters a keyword. For example, the following String macro cannot be called because a Word macro removes the word CONTAINS:

```
WD     CONTAINS :
SD     BLOCK CONTAINS &1
```

- Improper specification of the source words within a COBOL NOTE, comment, or literal.
- The appropriate COBOL division header does not precede the source words.
- A MetaCOBOL+ reserved word has been improperly specified as a qualified symbolic operand. For example, the following MOVE macro can not be called because KEY is a COBOL reserved word and therefore can not be an identifier or literal:

```
SP      MOVE &1 (S,L) TO &2(S)
           ...
        MOVE KEY TO REF-CODE
```

Improper Area Alignment

Improper Area A word alignment is generally caused by:

- A source or model substitution word inadvertently beginning in column 8-11.
- An &A directive being inadvertently executed in a macro model.

Improper formatting of an Area A word within Area B is generally caused by:

- Source and model substitution words which do not force Area A alignment.
- Improper execution of an &B directive.
- Inadvertent &STOW or &STORE of an Area A indicator (i.e., &N'T='A').

Incomplete Substitution

Incomplete word substitution via a macro model is generally caused by:

- Failure to include constant or symbolic substitution words in the macro model.
- Improper transfer of control bypassing model substitution words.
- Substitution to the wrong output location, indicating a logical error in out-of-line substitution.
- Out-of-line substitution to a non-existent location, resulting in generation of the Lost Out-of-Line listing (described in the *Macro Facility Reference*.)
- Incomplete translation of the "logical" macro prototype (i.e., the String-macro prototype AND all words to be acquired via &GET/&STOW/&STORE logic). Normally, a portion of the "logical" prototype remains in the generated COBOL. This situation is usually caused by improper and premature determination of the logical terminating word.

Generated COBOL Out of Order

Whenever major portions of the generated COBOL program are out of order, the cause generally is:

- Out-of-line substitution remains active following macro model execution due to an executed &NOEND directive. No subsequent macro call has the &NOEND. Refer to the *Macro Facility Reference.*

Improper Terminating Period Substitution

Substitution of an unnecessary and unwanted period in the COBOL output is generally caused by:

- An unnecessary period following a source statement.
- An inadvertent period following a constant or symbolic substitution word.
- An inadvertent period following a directive or directive expression. For example, the following &SET directive expression does not include the period, which is translated as a substitution word:

```
&SET &VX = &VX + 1.
```

Consecutive Period Substitution

Consecutive terminating periods in the generated COBOL output are generally caused by:

- Improper substitution of a concatenation construction or variable containing a terminating period, resulting in loss of identity of the period as a separate word.

Garbled Word Substitution

Incomprehensible or garbled substitution words in the generated output are generally caused by:

- An improperly specified concatenation construction.
- Improperly initialized symbolic substitution words referenced in a concatenation construction or substituted directly.
- Inadvertent reference to the wrong variable name in a line output record.

Improper Out-of-line Format

Improper out-of-line format is generally caused by:

- Failure to specify &ENV, &DATA, or &PROC where out-of-line substitution crosses division boundaries via an &POINT directive expression (e.g., &DATA, &PONT).
- Substitution via &ANTE, &AUX, or &POST of other than line output records.

Lost Out-of-Line Words

Out-of-line substitution words appearing in the Lost Out-of-Line listing are generally caused by:

- A missing division header in the source.
- A division header substitution out-of-line.
- Failure to drop an &MARKER.

CA-MetaCOBOL+ Translator in a Loop

Logical errors can cause the CA-MetaCOBOL+ Translator to loop continuously. This condition is generally caused by:

- A closed loop in a macro model, with no provisions for exit.
- A closed loop in a macro model, with improper testing of a counting variable. For example, the following &REPEAT loop will never exit if the current value of &1 cannot be found in the &VY table, because &VX will overflow from 9 to 1 and, therefore, never be greater than 9.

```
.   .   .
&LOCAL &VX 9
&LOCAL &VY(9) X(30)
&SET &VX = 0
&REPEAT
     &SET &VX = &VX + 1
&UNTIL &VX > 9
        OR &VY(&VX) NOT = &1
&ENDREP
.   .   .
```

User-Defined Macro Debugging Aids

Following logical analysis of test results, it is sometimes necessary to verify macro calls, current values of symbolic words, and other conditions which are difficult to debug at the source level.

The &NOTE directive and line output records are essential tools that the macro writer can use to verify macro calls.

Examples

Assume that a $-LEVEL macro does not appear to be processing certain data item definitions. The results displayed on the Input & Diagnostics listing indicate that the event which calls the $-LEVEL macro--a period preceding a numeric literal--is disabled following a group item definition. This indicates that the third word, if a period, is removed from the source, as illustrated below:

CA-MetaCOBOL+ Input Listing:

```
          SD     $-LEVEL  :
                    &GET &1
                    &STORE
                    &1
                    &NOTE &( 'LEVEL ' &1 ' CALLED' &)
           .   .   .
           DATA DIVISION.
           .   .   .
           WORKING-STORAGE SECTION.
           01 RECORD-A.
****NOTE N99 LEVEL 01 CALLED
                02 ITEM-02.
                   03 ELEM-03 PIC X.
****NOTE N99 LEVEL 03 CALLED
           .   .   .
```

The &NOTE directive can be used to display the current values of symbolic words for logical verification. For example, one might define a Word macro to dump the current contents of global variables when that word is used as a substitution word, as follows:

CA-MetaCOBOL+ Input Listing:

```
          WI     $$GLOBAL :
                     &GLOBAL &VA X(30)
                     &GLOBAL &VB 9(2)
          .  .  .
          W      @DEBUG:
                     &NOTE &( '&VA=' &VA &)
                     &NOTE &( '&VB=' &VB &)
          .  .  .
          W      DIVISION :
                     DIVISION
                     @DEBUG
           .  .  .
           IDENTIFICATION DIVISION
****NOTE N99 &VA=
****NOTE N99 &VB=0
           .  .  .
           ENVIRONMENT DIVISION.
****NOTE N99 &VA=XYZ
****NOTE N99 &VB=21
           .  .  .
```

When developing complex macro sets involving tables of interrelated variables, the development of such debugging aids should be pre-planned. Original test results will then provide debugging information to enhance logical analysis.

CA-MetaCOBOL+ Debugging Aid Facility

The macro debugging aid facility is an integral part of CA-MetaCOBOL+ and can be used to trace logical macro errors. In order to activate the debugging aid, the macro writer must define both the extent and level of logical tracing.

Defining Macros to be Traced

Two translator-directing statements must be inserted in the macro set to define the macros to be traced.

Format:

```
*$TRACE

*$NOTRACE
```

The *$TRACE command in columns 7-13 specifies that all subsequent macros are to be traced until detection of an *$NOTRACE command in columns 7-15. The *$TRACE and *$NOTRACE statements are ignored if the translate-time option TRACE= has not been specified.

The Translate-Time Option - TRACE=

Macro debugging can be invoked for all macros defined between *$TRACE and *$NOTRACE commands by specifying the TRACE= translate-time option.

```
       {D}
TRACE= {G}
       {S}
       {T}
```

TRACE=D

specifies that all "Nnn" class Translator diagnostics (as defined in the CA-MetaCOBOL+ *User Guide*), including NOTE and FLAG diagnostics, will display an additional message indicating the macro line number which caused the diagnostic, if the macro being executed is subject to tracing. The message format is:

```
####TRACE-    DIAGNOSTIC IN MACRO LINE #nnnn
```

TRACE=G

specifies that all source words acquired via an &GET directive expression will display a message indicating the macro line number containing the &GET, the Type attribute, and the word acquired, provided the macro being executed is subject to tracing. The format of the message is:

```
####TRACE-    &GET IN MACRO LINE #nnn 't' @word@
```

TRACE=S

specifies that all data items acquired via an $SCAN-type directive expression will display a message indicating the macro line number containing the directive and the data-name acquired, provided the macro being executed is subject to tracing. The format of the message is:

```
####TRACE-  SCANx IN MACRO LINE #nnnn @data-name@
```

TRACE=T

specifies that each executed transfer of control will display a message indicating that macro line number containing the transfer, provided the macro being executed is subject to tracing. The format of the message is:

```
####TRACE-    TRANSFER IN MACRO LINE #nnnn
```

The operands of the TRACE= option can be specified in combination and in any order (e.g., TRACE=TGD).

Tracing Diagnostics - TRACE=D

CA-MetaCOBOL+ ERRORS, WARNINGs, NOTEs, and FLAGs are numbered diagnostics, as listed in the *CA-MetaCOBOL+ User Guide.* For example, all ERRORs, and WARNINGs representing macro syntax errors and displayed during Macro Loading are encoded "Cnn" where *nn* is the number of the diagnostic. Other classes of diagnostics are displayed during execution of the Parameter Analysis, Memory Management, Input Parsing, and Macro Translation modules. (These modules are explained in the *Macro Facility Reference.*)

Diagnostics encoded "Nnn", including NOTEs, are displayed during macro execution and represent conditions in macro logic. Since this category of diagnostics is displayed following a source statement, but actually represents a condition in a macro, it is often difficult to determine which macro statement caused the diagnostic during translation. The TRACE=D option is included to provide the necessary correlation between N-class diagnostics and the macro statements which cause them.

As an example of TRACE=D debugging, assume that a complex $-LEVEL macro produces the diagnostic "N04 NON-NUMERIC DATA IN &SET OR &IF" in the source Data Division. Logical analysis of the $-LEVEL macro cannot isolate the &SET or &IF directive expression which has caused the error. The following example uses the TRACE=D option to determine the macro statement number:

CA-MetaCOBOL+ Input Listing:

```
          0050         *$TRACE
          0051         SD    $-LEVEL :
                             .   .   .
          0090               &SET &VX - &VY + 1
              .   .   .
          0120         *$NOTRACE
                        .   .   .
          0400          DATA DIVISION.
          0401          WORKING-STORAGE SECTION.
          0402          77 R-COUNT    PIC S9(5)V99 COMP-3 VALUE
+0.
          ****  ERROR   N04 NON-NUMERIC DATA IN &SET OR &IF
          ####  TRACE-   DIAGNOSTIC IN MACRO LINE #0090
                        .   .   .
```

The preceding example indicates that the error was caused by execution of line 0090. Either &VY contains a non-numeric current value or &VX is defined as non-numeric.

&GET Tracing - TRACE=G

Continued source acquisition via &GET and &STOW/&STORE directives in String macro models is often difficult to debug. The primary reason for this difficulty is that the words passed to Macro Translation from Input Parsing are not readily apparent from an examination of the source statement. Remember that Word and Prefix macros can be expanded when acquired via the &GET, and that certain control words, such as the Area A indicator, can be detected (refer to Chapter 9). Improper termination testing can result in a premature exit from the macro or a runaway &GET/&STOW loop which continues through end-of-input and produces the "H91 READ PAST END OF INPUT" diagnostic as a FATAL ERROR.

The TRACE=G option is a convenient method for examining the actual words obtained via the &GET directive expression. The TRACE=G option can be used to isolate the cause of improper &GET and &STOW/&STORE termination, inability to call a String macro, or a runaway source acquisition loop.

As an example, assume that a BLOCK CONTAINS String macro is NOT called in a complex macro set. An FD macro can be written to determine the cause and to ignore all other String macro calls, as follows:

CA-MetaCOBOL+ Input Listing:

```
0010        WD     CONTAINS  :
                   .  .  .
0400        SD     BLOCK CONTAINS &1 RECORDS :
                   .  .  .
0600        *$TRACE
0601        SD     FD :
0602               FD
0603                    &REPEAT
0604                         &GET &1
0605                         &STOW
0606                         &1
0607                    &UNTIL &1'T = 'V' &AND &1 = '.'
0608                    &ENDREP
0609        *$NOTRACE
             .  .  .
0900         DATA DIVISION.
0901         FILE SECTION.
0902         FD   MASTER-FILE
####  TRACE-   &GET IN MACRO LINE #0604 ' ' @MASTER-FILE@
0903               BLOCK CONTAINS 10 RECORDS
####  TRACE-   &GET IN MACRO LINE #0604 'S' @BLOCK@
####  TRACE-   &GET IN MACRO LINE #0604 'L' @10@
####  TRACE-   &GET IN MACRO LINE #0604 ' ' @RECORDS@
```

Notice that the type attribute of the word BLOCK is 'S' for a String macro name, instead of 'V' for a Data Division verb, and that the word CONTAINS is not acquired, since the Word macro on line 00010 removes it. The BLOCK CONTAINS macro, therefore, cannot be called.

Tracing Scans - TRACE=S

Data item acquisition via &SCAN-type directives often requires visible verification of the data-names returned in &0. Such verification is easily handled by inserting an &NOTE directive immediately following each &SCAN-type directive in the macro set, as follows:

```
...
&REPEAT
   &SCAN &1 &1
&UNTIL ENDSCAN
   &NOTE &( 'OPERAND &0 CONTAINS ' &0 &)
&ENDREP
...
```

The disadvantage of this approach, however, is that the &NOTE directives are for verification only, and must eventually be removed.

The TRACE=S option is a more convenient technique for accomplishing the same function, as shown in the following example.

CA-MetaCOBOL+ Input Listing:

```
         0499        *$TRACE
         0500        SP     $PDE :
         0501               &REPEAT
         0502                     &SCAN WORKING-STORAGE
WORKING-STORAGE
         0503               &UNTIL ENDSCAN
                                 .   .   .
         0600               &ENDREP
                      .   .   .
         0900         DATA DIVISION.
         0901         WORKING-STORAGE SECTION.
         0902         01 COUNTERS USAGE COMP-3.
         0903            02 COUNTER-1 PIC S9(5) VALUE +0.
                      .   .   .
         1000         PROCEDURE DIVISION.
                      .   .   .
         ####  TRACE-&SCAN IN MACRO LINE #502 @WORKING-STORAGE@
         ####  TRACE-&SCAN IN MACRO LINE #502 @COUNTERS@
```

Tracing Transfers of Control - TRACE=T

Logical transfers of control within macro models are often difficult to analyze and debug, especially if the CA-MetaCOBOL+ Translator is looping continuously in an undetermined macro model. The TRACE=T option is the most convenient technique for determining the location of such logical errors.

Assume that CA-MetaCOBOL+ is looping continuously during translation of the Data Division. The following example indicates a closed &REPEAT loop in the $-LEVEL macro. The variable &VX defined in line 401 is not large enough to handle a value greater than 9.

CA-MetaCOBOL+ Input Listing:

```
        0300        *$TRACE
                    .  .  .
        0400        SD     $-LEVEL :
                           &LOCAL &VX 9
                           .  .  .
        0459               &REPEAT
        0460                    &SET &VX = &VX + 1
        0461               &UNTIL &VX > 9
        0462               &ENDREP
                    .  .  .
        0600        *$NOTRACE
                     .  .  .
        0700         DATA DIVISION.
        0701         WORKING-STORAGE SECTION.
        0702         77    ITEM -A               PIC X.
        ####  TRACE-       TRANSFER IN MACRO LINE #0462
        ####  TRACE-       TRANSFER IN MACRO LINE #0462
        ####  TRACE-       TRANSFER IN MACRO LINE #0462
        .  .  .

                    (previous line repeats until job is
cancelled)
```

The CA-MetaCOBOL+ Procedure Documentor (MPD)

The CA-MetaCOBOL+ Procedure Documentor is a COBOL program that is delivered with the various macro sets at installation time.

Its function is to list one or more macro sets, assign a unique sequence number to each line, and produce various cross-reference reports.

This facility is described in detail in the *Macro Facility Reference*.

12.3 Summary - Standards and Debugging

Listed below are the key points covered in this section.

- The following standard organization is recommended for sets of macros:

  1. Prologue.
  2. External and Global variable definitions.
  3. Macros organized in standard COBOL division sequence.

- Use standard macro formats and generate standard prefixed data-names, suffixed procedure-names, and prefixed diagnostics.
- The Input and Diagnostics listing Statistics section shows CA-MetaCOBOL+ errors and warnings.
- The &NOTE directive can be used to generate debugging diagnostics and check points.
- The CA-MetaCOBOL+ Debugging Aid Facility provides the statements *$TRACE and *$NOTRACE.
- The TRACE=translate-time option must be specified with them.
- The CA-MetaCOBOL+ Procedure Documentor lists macro sets, providing sequence numbers and cross references.

Appendix A - Solutions to Exercises

This appendix contains suggested solutions to each of the exercises in this volume. Alternate solutions may, of course, be quite acceptable.

Exercise 1: Define short forms to generate the data-names BALANCE-ON-HAND, ISSUES-TODAY, RECEIPTS-TODAY, and PREVIOUS-BALANCE, and the paragraph-name COMPUTE-NEW-BALANCE.

CA-MetaCOBOL+ Input:

```
WDP    BOH : BALANCE-ON-HAND
WDP    IST : ISSUES-TODAY
WDP    RCT : RECEIPTS-TODAY
WDP    PRB : PREVIOUS-BALANCE
WP     CNB : COMPUTE-NEW-BALANCE
 .  .  .
 &DD
 WORKING-STORAGE SECTION.
 77 BOH PIC S9(7)V99 COMP-3 VALUE +0.
 77 IST PIC S9(7)V99 COMP-3 VALUE +0.
 77 RCT PIC S9(7)V99 COMP-3 VALUE +0.
 77 PRB PIC S9(7) V99 COMP-3 VALUE +0.
 .  .  .
 $PD
 .  .  .
      GO TO CNB.
 .  .  .
 CNB.  COMPUTE BOH = PRB - IST + RCT.
 .  .  .
```

CA-MetaCOBOL+ Output:

```
         .  .  .
         DATA DIVISION.
         WORKING-STORAGE SECTION.
         77  BALANCE-ON-HAND                PIC S9(7)V99 COMP-3
VALUE +0.
         77  ISSUES-TODAY            PIC S9(7)V99 COMP-3 VALUE
+0.
         77  RECEIPTS-TODAY          PIC S9(7)V99 COMP-3 VALUE
+0.
         77  PREVIOUS-BALANCE               PIC S9(7)V99 COMP-3
VALUE +0.
         .  .  .
         PROCEDURE DIVISION.
         .  .  .
             GO TO COMPUTE-NEW-BALANCE.
         .  .  .
         COMPUTE-NEW-BALANCE.
             COMPUTE BALANCE-ON-HAND = PREVIOUS-BALANCE -
ISSUES-TODAY
                   + RECEIPTS-TODAY.
         .  .  .
```

Exercise 2: Write Word and Prefix macros to simplify the underlined code listed below.

```
DATA DIVISION.
FILE SECTION.
FD OLD-MASTER-FILE  LABEL RECORDS ARE STANDARD
                    RECORDING MODE F
                    BLOCK CONTAINS 0 CHARACTERS
                    RECORD CONTAINS 80 CHARACTERS
                    DATA RECORD IS OLD-MASTER-RECORD.

01 OLD-MASTER-RECORD.
                    02 OLD-PLANT PIC 9(002).
                    02 OLD-DEPARTMENT  PIC 9(002).
                    02 OLD-ACCOUNT     PIC X(004).
                    02 OLD-AMOUNTPIC S9(05.  COMP-3.
                    02 OLD-DESCRIPTION PIC X(025).
                    02 OLD-SUFFIXPIX X(010).
                    02 FILLER    PIC X(052).
```

CA-MetaCOBOL+ Input:

```
PD     RECFM=       :  RECORDING MODE &
PD     BLKSIZE=     :  BLOCK CONTAINS & CHARACTERS
PD     LRECL=       :  RECORD CONTAINS & CHARACTERS
WD     LABELS       :  LABEL RECORDS ARE STANDARD
PD     RECORD=      :  DATA RECORD IS &.  01 &.
PD     ITEM=        :  02 &
PD     NUMERIC=     :  PIC &( 9( & ) &).
PD     CHARACTER=   :  PIC &( X( & ) &).
PD     PACKED=      :  PIC &( S9( & ) &) COMP-3.
PD     SLACK=       :  02 FILLER PIC &( X( & ) &).
 .  .  .
 $DD
 FILE SECTION.
 FD OLD-MASTER-FILE
        RECFM=F BLKSIZE=0 LRECL=80 LABELS
        RECORD=OLD-MASTER-RECORD
          ITEM=OLD-PLANT              NUMERIC=2
          ITEM=OLD-DEPARTMENT         NUMERIC=2
          ITEM=OLD-ACCOUNT            CHARACTER=4
          ITEM=OLD-AMOUNT             PACKED=5
          ITEM=OLD-DESCRIPTION        CHARACTER=25
          ITEM=OLD-SUFFIX             CHARACTER=10
          SLACK=52
 .  .  .
```

CA-MetaCOBOL+ Output:

```
        .  .  .
        DATA DIVISION.
        FILE SECTION.
        FD    OLD-MASTER-FILE
                                    RECORDING MODE F
                                    BLOCK CONTAINS 0 CHARACTERS
                                    RECORD CONTAINS 80 CHARACTERS
                                    LABEL RECORDS ARE STANDARD
                                    DATA RECORD IS
OLD-MASTER-RECORD.
        01    OLD-MASTER-RECORD.
              02 OLD-PLANT          PIC 9(2)
              02 OLD-DEPARTMENT     PIC 9(2)
              02 OLD-ACCOUNT        PIC X(4).
              02 OLD-AMOUNT         PIC S9(5) COMP-3.
              02 OLD-DESCRIPTION    PIC X(25).
              02 OLD-SUFFIX         PIC X(10).
              02 FILLER             PIC X(52).
        .  .  .
```

Exercise 3: The COBOL SEARCH verb does not initialize the index into the table to be searched. Write a String macro that finds each occurrence of,

```
SEARCH EVERY table-item index-name
```

where table-item is defined with an OCCURS clause, and index-name is the name of the first index defined by the item. The macro model initializes the index to 1 and outputs a SEARCH table-item statement.

```
SET index-name TO 1
SEARCH table-item
```

As a second exercise, rewrite the FD entries in Exercise 2 using a String macro. (Remember, you can nest Word macros within the String macro.)

I.

CA-MetaCOBOL+ Input:

```
SP     SEARCH EVERY &1(Q) &2  :
       SET &2 TO 1
       SEARCH &1
 .  .  .
 $PD
       SEARCH EVERY PLANT-ENTRY PLANT-INDEX
       .  .  .
```

CA-MetaCOBOL+ Output:

```
.  .  .
PROCEDURE DIVISION.
      SET PLANT-INDEX TO 1
      SEARCH PLANT-ENTRY
      .  .  .
```

II.

CA-MetaCOBOL+ Input:

```
SD     FILE= &1 &2 &3 &4 &5 :
                      FD &1
                      RECORDING MODE &2
                      BLOCK CONTAINS &3 CHARACTERS
                      RECORD CONTAINS &4 CHARACTERS
                      LABELS
                      DRI &5
WD     DRI         : DATA RECORD IS
WD     LABELS      : LABEL RECORDS ARE STANDARD
 .  .  .
 $DD
 FILE SECTION.
       FILE= OLD-MASTER-FILE F 0 80 OLD-MASTER-RECORD.
                  .  .  .
```

CA-MetaCOBOL+ Output:

```
.  .  .
DATA DIVISION.
FILE SECTION.
FD     OLD-MASTER-FILE
                      RECORDING MODE F
                      BLOCK CONTAINS 0 CHARACTERS
                      RECORD CONTAINS 80 CHARACTERS
                      LABEL RECORDS ARE STANDARD
                      DATA RECORD IS OLD-MASTER-RECORD.
.  .  .
```

Exercise 4: Define two macros to generate the following:

INPUT-OUTPUT SECTION.
FILE-CONTROL.

WORKING-STORAGE SECTION.

CA-MetaCOBOL+ Input:

```
WE     -IOFC    :
                  &A INPUT-OUTPUT SECTION.
                  &A FILE-CONTROL.
WD     -WS   :
 WORKING-STORAGE SECTION.
 .  .  .
 $ED
 .  .  .
 -IOFC
 .  .  .
 $DD -WS
 .  .  .
```

CA-MetaCOBOL+ Output:

```
.  .  .
ENVIRONMENT DIVISION.
.  .  .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
.  .  .
DATA DIVISION.
WORKING-STORAGE SECTION.
.  .  .
```

Exercise 5: Write a String macro to diagnose each ALTER verb found in the Procedure Division with a single diagnostic that contains the altered procedure-name, as:

```
procedure-name ALTERED - STANDARDS ERROR
```

Remember, the &NOTE directive expression can include a concatenation construction.

CA-MetaCOBOL+ Input:

```
          SP     ALTER &1(Q) :
                     ALTER &1
                     &NOTE
                       &( &1 ' ALTERED - STANDARDS ERROR' &)
           .  .  .
           PROCEDURE DIVISION.
           .  .  .
                 ALTER PARA-A TO ...
****NOTE N99 PARA-A ALTERED - STANDARDS ERROR
                 .  .  .
                 ALTER PARA-C TO ...
****NOTE N99 PARA-C ALTERED - STANDARDS ERROR
```

To create a single diagnostic message containing both constant and variable information, combine them into a single word by means of a concatenation.

Notice that within the concatenation, the literal contains embedded spaces. Concatenation removes the quotation marks but retains these spaces so that the words in the message are readable as normal text.

Exercise 6: Define a CA-MetaCOBOL+ String macro that will store file names taken from Environment Division SELECTs in a table. The table must be available for subsequent reference by other macros in the region. It must be large enough to hold up to nine file names.

CA-MetaCOBOL+ input might appear as:

```
. . .
ENVIRONMENT DIVISION.
FILE-CONTROL
SELECT INPUT-FILE ASSIGN TO...
SELECT MASTER-FILE ASSIGN TO...
SELECT REPORT-FILE ASSIGN TO...
. . .
```

Your macro should build a table containing the following entries:

```
INPUT-FILE
MASTER-FILE
REPORT-FILE
. . .
```

Note: The SELECT statements must appear in the output unaltered.

CA-MetaCOBOL+ Input:

```
          SE     SELECT &1 :
                 SELECT &1
                 &GLOBAL &VFILENAMES(9) X(30)     /* FILENAME
TABLE
                 &LOCAL &VINDEX         9(02)     /* TABLE INDEX
                 &SET &VINDEX = &VINDEX + 1 /* BUMP INDEX
                 &SET &VFILENAMES(&VINDEX) = &1   /* STORE
FILENAME
           .  .  .
           $ED
           .  .  .
           FILE-CONTROL.
                 SELECT INPUT-FILE ASSIGN TO INPUT.
                 SELECT MASTER-FILE ASSIGN TO MASTER.
                 SELECT REPORT-FILE ASSIGN TO REPORT.
           .  .  .
```

CA-MetaCOBOL+ Output:

```
.   .   .
ENVIRONMENT DIVISION.
.   .   .
FILE-CONTROL.
      SELECT INPUT-FILE
                          ASSIGN TO INPUT.
      SELECT MASTER-FILE
                          ASSIGN TO MASTER.
      SELECT REPORT-FILE
                          ASSIGN TO REPORT.
.   .   .
```

Exercise 7: Given an indexed global variable defined as,

```
&GLOBAL &VFILE(9) X(30)
```

and containing up to nine file names, write a Word macro that replaces CLOSEALL with:

```
CLOSE file-name-1 file-name-2...
```

CA-MetaCOBOL+ input might appear as follows:

```
.   .   .
ENVIRONMENT DIVISION.
.   .   .
   SELECT INPUT-FILE  ASSIGN TO...
   SELECT MASTER-FILE ASSIGN TO...
   SELECT REPORT-FILE ASSIGN TO...
   .   .   .
PROCEDURE DIVISION.
.   .   .
   CLOSEALL.
   .   .   .
```

Procedure Division translation should be:

```
.   .   .
PROCEDURE DIVISION
.   .   .
   CLOSE INPUT-FILE MASTER-FILE REPORT-FILE.
   .   .   .
```

Remember to reinitialize the required variables, in case more than one CLOSEALL is specified. Do not generate a CLOSE statement if the table is empty.

CA-MetaCOBOL+ Input:

```
    SE     SELECT &1 :
           SELECT &1
           &GLOBAL &VFILE(9) X(30)          /* FILE-NAME TABLE
           &LOCAL &VX = 0                   /* FILE INDEX
           &SET &VX = &VX + 1
           &IF &VX LT 10                    /* IF < 10 FILE-NAMES
                &SET &VFILE(&VX) = &1       /* SAVE IN TABLE
           &ELSE                            [/* IF > 10 NOTE]
                &NOTE &( 'FILE NAME ' &1 ' NOT INCLUDED' &)
           &ENDIF
    WP     CLOSEALL :
           &LOCAL &VX = 0 99
           &IF &VFILE(1) NE ' '             /* IF TABLE NOT EMPTY
                CLOSE &VFILE(1)             /*  CLOSE 1ST NAME
           &ENDIF
           &SET &VX = 2
           &REPEAT                          /* FOR REST OF TABLE
           &UNTIL &VFILE(&VX) EQ ' '        /*  UNTIL BLANK ENTRY
                &OR &VX GT 9                /*  OR > 9 ENTRIES
                &VFILE(&VX)                 /*  OUTPUT FILE-NAME
                &SET &VX = &VX + 1          /*  BUMP INDEX
           &ENDREP
     .  .  .
     ENVIRONMENT DIVISION.
     .  .  .
     FILE-CONTROL.
           SELECT FILE1 ASSIGN TO DD1.
           SELECT FILE2 ASSIGN TO DD2
           SELECT FILE3 ASSIGN TO DD3.
           .  .  .
     PROCEDURE DIVISION.
     .  .  .
           CLOSEALL.
```

CA-MetaCOBOL+ Output:

```
.  .  .
ENVIRONMENT DIVISION.
.  .  .
FILE-CONTROL.
     SELECT FILE1
                          ASSIGN TO DD1.
     SELECT FILE2
                          ASSIGN TO DD2.
     SELECT FILE3
                          ASSIGN TO DD3.
     .  .  .
PROCEDURE DIVISION.
.  .  .
     CLOSE FILE1 FILE2 FILE3.
```

Exercise 8: Write a macro to interrogate each MOVE statement in a COBOL program. If the receiving field of a MOVE statement contains more than 256 bytes, print the diagnostic:

```
data-name IS LARGER THAN 256 BYTES
```

CA-MetaCOBOL+ Input:

```
          SP     MOVE &1(S,L) TO &2(S) : MOVE &1 TO &2
                 &IF &2'S > 256
                    &NOTE &( &2 ' IS LARGER THAN 256 BYTES.' &)
                 &ENDIF
          .  .  .
          $DD
          WORKING-STORAGE SECTION.
          77     SHORT-DATA            PIC X(256).
          77     LONG-DATA             PIC X(257).
          $PD
                       MOVE SPACES TO LONG-DATA.
**** NOTE N99          LONG-DATA IS LARGER THAN 256 BYTES.
                       MOVE SPACES TO SHORT-DATA.
```

CA-MetaCOBOL+ Output:

```
          .  .  .
          DATA DIVISION.
          WORKING-STORAGE SECTION.
          77     SHORT-DATA            PIC X(256).
          77     LONG-DATA             PIC X(257).
          PROCEDURE DIVISION.
                 MOVE SPACES TO LONG-DATA.
                 MOVE SPACES TO SHORT DATA.
```

Exercise 9: An OPEN-PRINT macro was previously defined to OPEN the printer file and then pass the record-name, depth-of-page, and name of the page-heading-routine to the PRINT macro. For each of reference, this macro is shown:

```
SP   OPEN-PRINT &1 USING &2 DEPTH &3(L)
                    OVERFLOW &4 :
           OPEN OUTPUT &1
           PERFORM &4.
           &GLOBAL &VRECORD X(30)
           &GLOBAL &VDEPTH 9(2)
           &GLOBAL &VOFLO X(30)
           &SET &VRECORD =&2
           &SET &VDEPTH = &3
           &SET &VOFLO =&4
```

The "OPEN-PRINT file-name USING record-name" portion in the prototype was required because the OPEN statement needs the file-name, and the WRITE statement requires the record-name.

Given the file-name, it is relatively easy to find the record-name by looking for the first 01 level entry after the printer FD. Modify the above macro so that the syntax of the OPEN-PRINT macro prototype reads:

```
OPEN-PRINT &1 DEPTH &2(L) OVERFLOW &3
```

CA-MetaCOBOL+ Input:

```
     SP     OPEN-PRINT &1 DEPTH &2(L) OVERFLOW &3 :
            OPEN OUTPUT &1
            PERFORM &3.
            &GLOBAL &VRECORD X(30)          /* RECORD NAME
            &GLOBAL &VDEPTH 99              /* PAGE DEPTH
            &GLOBAL &VOFLO X(30)            /* PAGE HEADING
            &SET &VDEPTH = &2
            &SET &VOFLO = &3
            &SET &VRECORD = NULL
            &REPEAT                         /* SCAN FILE-NAMES
                 &SCAN &1 &1                /*
            &UNTIL ENDSCAN                  /* UNTIL DONE
                 &IF &0'L EQ '01'           /* IF LEVEL 01
                      &SET &VRECORD = &0    /*  SAVE RECORD NAME
                      &SCANX                /*  END SCAN
                      &ESCAPE               /*  EXIT
                 &ENDIF
            &ENDREP
            &IF &VRECORD EQ NULL            /* NOTE IF NO LEVEL
01
                 &NOTE &( &1 ' - INVALID FILE NAME' &)
            &ENDIF
     SP     PRINT &1(S) BY &2(L) :
            ADD &2 TO LINE-COUNT.
            IF LINE-COUNT > &VDEPTH
                 PERFORM &VOFLO.
            WRITE &VRECORD FROM &1
                 AFTER ADVANCING &2 LINES.
      DATA DIVISION.
      FILE SECTION.
      FD    LIST-FILE.
      01    LIST-RECORD
      PROCEDURE DIVISION.
            OPEN-PRINT LIST-FILE DEPTH 60 OVERFLOW NEW-PAGE-RTN.
            PRINT ERROR-LINE BY 1.
```

CA-MetaCOBOL+ Output:

```
.   .   .
DATA DIVISION.
FILE SECTION.
FD     LIST-FILE.
01     LIST-RECORD.
PROCEDURE DIVISION.
       OPEN OUTPUT LIST-FILE
       PERFORM NEW-PAGE-RTN.
       ADD 1 TO LINE-COUNT.
       IF LINE-COUNT > 60
            PERFORM NEW-PAGE-RTN.
       WRITE LIST-RECORD FROM ERROR-LINE
            AFTER ADVANCING 1 LINES.
```

Exercise 10: COBOL programmers must often define a Working-Storage area containing specific values, and then REDEFINE the area as a table. For example:

```
01 TABLE-AREA-1.
   02 FILLER         PIC X(14) VALUE 'ALABAMA'.
   02 FILLER         PIC X(14) VALUE 'ALASKA'.
   .  .  .
   02 FILLER         PIC X(14) VALUE 'WYOMING'.
01 TABLE-1           REDEFINES TABLE-AREA-1.
   02 STATE          OCCURS 50 TIMES
                     INDEXED BY STATE-INDEX
                     PIC X(14).
```

Write a macro that will create a table similar to this sample from the following input:

```
$TABLE 1 STATESTATE-INDEX
X(14)'ALABAMA''ALASKA'...'WYOMING'.
```

The macro prototype can be generalized as:

```
$TABLE level-number entry-name index-name
picture-string literal-1...literal-n.
```

Three variables are required to:

1. Count the calls to the $TABLE macro.

2. Count the literals representing values. The count becomes the object of the OCCURS clause.

3. Compute the starting level-number plus 1.

Obtain and count values until a non-literal word is encountered in the source.

CA-MetaCOBOL+ Input:

```
    SD     $TABLE &1(L) &2 &3 &4 :
           &LOCAL &V-TABLE 9(2)            /* $TABLE CALL COUNT
           &LOCAL &V-OCCURS 9(3)           /* OCCURS COUNT
           &LOCAL &V-LEVEL 9(2)            /* LEVEL + 1
           &SET &V-TABLE = &V-TABLE + 1
           &SET &V-OCCURS = 0
           &SET &V-LEVEL = &1 + 1
           &1 &( TABLE-AREA- &V-TABLE &).  /* OUT TABLE HEADER
           &REPEAT
               &GET &5                     /* GET NEXT WORD
           &UNTIL &5'T NOT = 'L'           /* UNTIL NOT LITERAL
               &STORE                      /* REMOVE WORD
               &V-LEVEL FILLER PIC &4 VALUE &5.  /* OUT FILLER ITEMS
               &SET &V-OCCURS = &V-OCCURS + 1
           &ENDREP
           &1 &( TABLE- &V-TABLE &) REDEFINES     /* OUT REDEFINES
               &( TABLE-AREA- &V-TABLE &).
           &V-LEVEL &2 OCCURS &V-OCCURS TIMES     /* OUT TABLE
ENTRIES
               &B INDEXED BY &3
               &B PIC &4.
     .  .  .
     $DD
     WORKING-STORAGE SECTION.
           $TABLE 1 STATE STATE-INDEX X(14) 'ALABAMA' 'ALASKA'
               ...  'WYOMING'.
```

CA-MetaCOBOL+ Output:

```
     .  .  .
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01     TABLE-AREA-1.
           02 FILLER              PIC X(14) VALUE 'ALABAMA'.
           02 FILLER              PIC X(14) VALUE 'ALASKA'.
        .  .  .
           02 FILLER              PIC X(14) VALUE 'WYOMING'.
    01     TABLE-1                REDEFINES TABLE-AREA-1.
           02 STATE               OCCURS 3 TIMES
               INDEXED BY STATE-INDEX
               PIC X(14).
     .  .  .
```

Exercise 11: Write a String macro which can be executed during translation of any COBOL division to relocate any statement coded as,

$77 data-name PICTURE picture VALUE value.

to the beginning of the Working-Storage section as the level-77 data item:

77 data-name PICTURE picture VALUE value.

CA-MetaCOBOL+ Input:

```
S      $77 &1 PICTURE &2 VALUE &3(L).   :
       &DATAWS
       77 &1 PICTURE &2 VALUE &3.
       &END
 .  .  .
 $DD
 WORKING-STORAGE SECTION.
 $PD
       IF SWITCH-5-17 = '8'
              $77 SWITCH-5-17 PICTURE X VALUE '7'.
              GO TO TMBTP-7-42.
```

CA-MetaCOBOL+ Output:

```
.  .  .
DATA DIVISION.
WORKING-STORAGE SECTION.
77     SWITCH-5-17            PICTURE X VALUE '7'.
PROCEDURE DIVISION.
       IF SWITCH-5-17 = '8'
         GO TO TMBTP-7-42.
```

Exercise 12: Today's compilers are more tolerant than in the past and this sometimes results in misplaced statements in the Procedure Division. Write a macro that will move verbs coded in Area A back into Area B.

CA-MetaCOBOL+ Input:

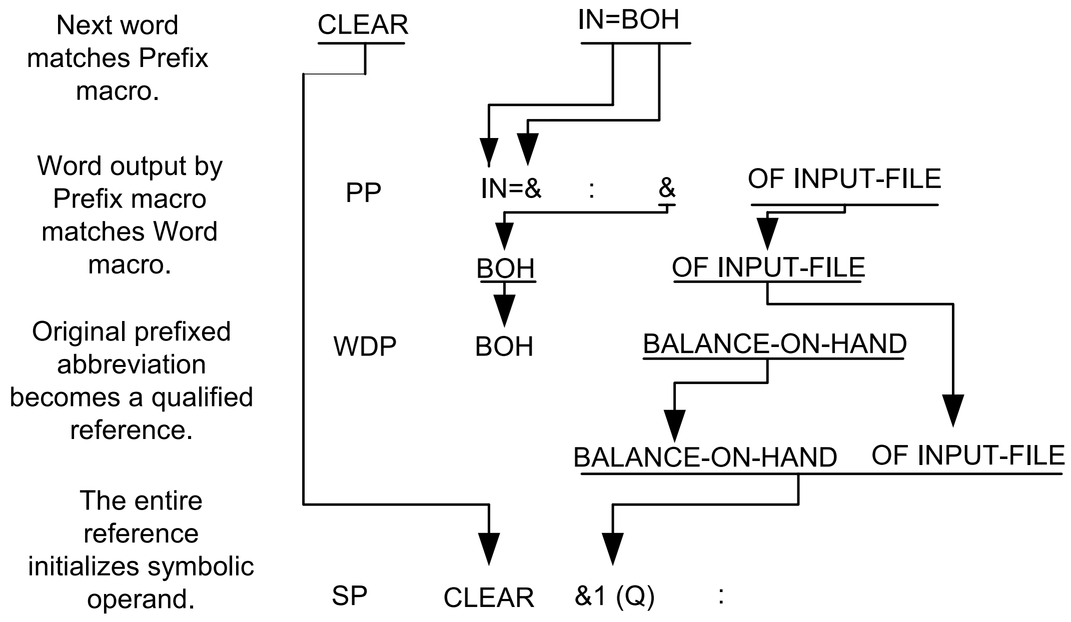
```
SP     $-PROC  :
       &GET &1
       &IF &1'T = 'V'
           &B
       &ENDIF
 .  .  .
 $PD
 MAIN SECTION.
 BEGIN.
            OPEN INPUT INPUT-FILE OUTPUT OUTPUT-FILE.
    PERFORM PAGE-HEADING-ROUTINE.
 READ-INPUT-FILE.
```

CA-MetaCOBOL+ Output:

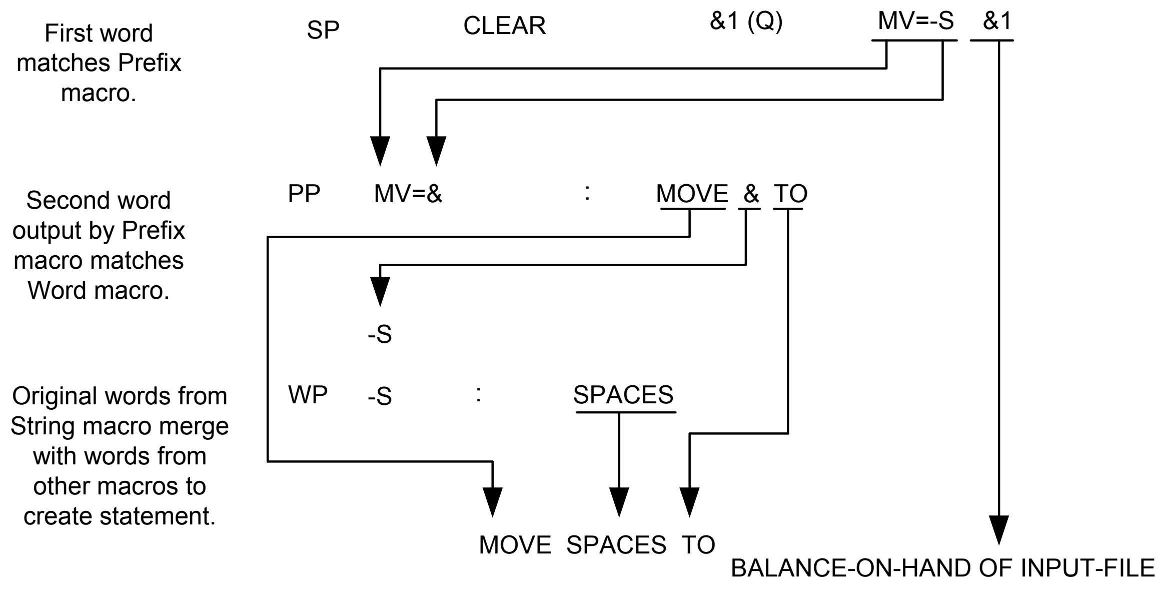
```
.  .  .
PROCEDURE DIVISION.
MAIN SECTION.
BEGIN.
    OPEN INPUT INPUT-FILE OUTPUT OUTPUT-FILE.
    PERFORM PAGE-HEADING-ROUTINE.
READ-INPUT-FILE.
```

Appendix B - Macro Nesting Illustrations

Procedure Division source text word CLEAR first matches String-macro name.


Next word matches Prefix macro.
Word output by Prefix macro matches Word macro.
Original prefixed abbreviation becomes a qualified reference.
The entire reference initializes symbolic operand.
CLEAR
IN=BOH
PP
IN=&
:
&
OF INPUT-FILE
BOH
OF INPUT-FILE
WDP
BOH
BALANCE-ON-HAND
BALANCE-ON-HAND
OF INPUT-FILE
SP
CLEAR
&1 (Q)
:

String macro substitution text consists of two words.


First word matches Prefix macro.
SP
CLEAR
&1 (Q)
MV=-S
&1
Second word output by Prefix macro matches Word macro.
PP
MV=&
:
MOVE
&
TO
-S
Original words from String macro merge with words from other macros to create statement.
WP
-S
:
SPACES
MOVE SPACES TO
BALANCE-ON-HAND OF INPUT-FILE

Appendix C - Symbolic Operand Attributes

Name/Notation	PIC	Description
Address &n'A	9(5)	Numeric value representing relative position of data item in TDS.
Displacement &n'D	9(5)	Numeric value representing the location of the left-hand byte of the data item relative to 0 as the beginning of the record. Level-01 and level-77 items always have displacement of 0.
Display Size &n'9	9(5)	Numeric value representing the display size of a numeric literal or data item; all other items return a value of 0.
EXTERNAL &n'E	X	Non-numeric value that is the character 'E' for an item specifying the external clause and for all items subordinate to a level 01 item specifying the external clause.
GLOBAL &n'G	X	Non-numeric value that is the character 'G' for an item specifying the global clause and for all items subordinate to an FD or level 01 item specifying the global clause.

NAME CHECK	X	returns a character value for the value of the symbolic operand. Shown below are the values and descriptions.

Value	Description
A	Ambiguous. The Attribute Table contains two or more sets of attributes for a data item in the Symbol Table but the attributes are imperfectly qualifed in the symbolic operand. This condition is "ambiguous."
L	Literal or Figurative Constant. The symbolic operand value is a literal or figurative constant.
Q	Unique Data Item. The symbolic operand value is a unique data item or a properly qualified duplicate data item name.
U	Undefined. The symbolic operand value is not in the Symbol Table or lacks attributes in the Attribute Table. It is undefined.

Level &n'L	X(2)	Non-numeric value representing data item level number, as follows:

\` '	DATA DIVISION section header.
\`00'	Level indicator (FD,CD,SD).
\`01'	Record description or mnemonic-name.
\`nn'	Level number of data item.
\`88'	DATA DIVISION condition-name.
\`89'	ENVIRONMENT DIVISION condition-name.
\`97'	COBOL special register.
\`98'	Index-name.

Name Size &n'N	9(3)	Numeric value representing the length of the first word in the symbolic operand.
OCCURS &n'O	9(5)	Numeric value representing the maximum value of the OCCURS clause for the data item (only on the item containing the OCCURS clause).
OCCURS Level &n'K	9	Numeric value representing the number of subscripts/indices required to reference the data item.

Name/Notation	**PIC**	**Description**
Point &n'P	99	Signed numeric value representing the decimal location in a numeric literal or data item, as follows: >0 Number of decimals. (S9V99 returns 2.) <0 Assumed number of positions ahead of the decimal point. (599PPP returns--3.) =0 No decimals or a non-numeric data item.
REDEFINES &n'R clause.	9	Numeric value indicating the presence or absence of a REDEFINES or RENAMES clause, as follows: 0 Not a REDEFINES or RENAMES data item, nor a data item subordinate to a REDEFINES 1 REDEFINES or RENAMES data item, or a data item subordinate to a REDEFINES clause.
Sign a &n'-	9	Numeric value representing the presence or absence of sign, as follows: 0 Unsigned or non-numeric. 1 Signed numeric literal or item. 2 Separately signed data item.
Size &n'S	9(8)	Numeric value representing the size of the field: Data items: Number of bytes required to contain the item. Numeric literals: Number of characters needed to contain the literal as written. Non-numeric literals: Length of the literal as written, including bounding quotes. Figurative constants: Size is 1.
SYNCHRONIZED &n'Y	9	Numeric value indicating the presence or absence of a SYNCHRONIZED clause, as follows: 0 Data item is not SYNCHRONIZED. 1 Data item is SYNCHRONIZED.

Type &n'T	X	Non-numeric value representing the syntax type of the current value of the symbolic operand, as follows: ` ' None of the following. `A' Next word begins in Area A (Area A indicator). `L' Word is a literal or figurative constant. `N' Word is a NOTE or comment. `S' Word matches a String-type macro name. `V' Word is a verb or terminating period.

Name/Notation	**PIC**	**Description**
USAGE &n'U	X	Non-numeric value representing the symbolic operand USAGE, as follows: `A' Alphabetic data item. `C' Condition-name. `F' Figurative constant. `G' Group data item. `I' Index data item. `J' Index-name. `M' Mnemonic-name. `N' Numeric literal. `Q' Non-numeric literal. `R' Report data item (edited). `U' Undefined or invalid. `X' Alphanumeric data item. `0' COMPUTATIONAL data item. `1' COMPUTATIONAL-1 data item. `2' COMPUTATIONAL-2 data item. `3' COMPUTATIONAL-3 data item. `4' External floating-point data item. `9' Numeric DISPLAY data item.
VALUE &n'V	9	Numeric value indicating the presence or absence of a VALUE clause, as follows: 0 No VALUE clause in this data item or a subordinate item. 1 VALUE clause in this data item or a subordinate item.

Appendix D - &SETR NOTE Register

To access the NOTE register, use the &SETR directive expression to define a variable to hold the returned value. (See Section 6.4.6.) The value returned by the NOTE register may be one of the following:

Register-name	Picture	Meaning		
NOTE	XX	NOTE type:		
		A	=	&ACCT
		B	=	Blank lines
		C	=	CICS statements (option PRESERVE=SQL must be specified)
		D	=	Debugging Line
		DP	=	&DSTOP
		DS	=	&DSTART
		E	=	ENABLED comment
		F	=	&A,&B,&INDENT,&OUTDENT
		L	=	Line output
		MK	=	&MARKER
		N	=	COBOL NOTE, REMARKS, comment, or statement subordinate to *$NOMCT
		S	=	Spacing controls (EJECT,SKIP1,SKIP2, SKIP3) or -INC
		0	=	&END
		1	=	&ANTE
		2	=	&ENV
		3	=	&DATAF
		4	=	&DATAWS
		5	=	&DATAW
		6	=	&DATAWX
		7	=	&DATAL
		8	=	&DATAR
		9	=	&DATA
		10	=	&PROCS
		11	=	&PROC
		12	=	&PROCX
		13	=	&POST
		14	=	&AUX

15	=	&POINT
18	=	&DUMMY

Appendix E - Glossary

The following terms and definitions are standard within CA-MetaCOBOL+:

Alphabetic character
: is a character of the set A through Z.

Area A
: Columns 8 through 11 within COBOL source format. The first word beginning in Area A is considered an Area A word in its entirety. Subsequent words beginning in Area A of the same record are considered Area B words.

Area A indicator
: An indicator passed through the CA-MetaCOBOL+ translation process to cause the subsequent output word to be placed in Area A.

Area B
: Columns 12-72 within COBOL source format. Any word, other than the first, physically beginning in Area A in a COBOL source record is considered an Area B word.

Comment lines
: Any line with an asterisk or slash in the continuation column (column 7). Such lines may be used to contain special codes or commands to control CA-MetaCOBOL+ input and output, or they may be used as American National Standards Institute (ANSI) COBOL comments.

Constructs
: Combinations of directives that define the logic and scope of selection and repetition directives.

Continuation column
: Column 7 of CA-MetaCOBOL+ input and of COBOL output statements.

Current Value
: The actual value of a symbolic word found in the input or assigned through translation.

Data item definition

A sentence in the DATA DIVISION beginning with a level number and ending with a period. A proper data item definition includes the name of the item and all necessary attribute clauses to describe the data element.

Data-name

A word, containing at least one alphabetic character, which defines a data item in the DATA DIVISION.

Directive

A special word, embedded within a CA-MetaCOBOL+ macro model, which causes the Translator to take explicit action but does not appear in the output. A directive is similar in function to COBOL verb which defines the action to be taken by a COBOL program. Directives always begin with an ampersand (&).

Directive expression

A series of words, beginning with a directive, which defines explicit Translator action. A directive expression is similar in function to a COBOL statement.

Division header

The words ID DIVISION, IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, PROCEDURE DIVISION, or PROCEDURE DIVISION USING..., beginning in Area A. CA-MetaCOBOL+ also recognized $ID, $ED, $DD, and $PD beginning in Area A as division headers.

Execute time

The run time, following compiling and link-editing, of a COBOL program after translation by CA-MetaCOBOL+.

Figurative constant

The fixed data-names ZERO, ZEROS, ZEROES, SPACE, SPACES, HIGH-VALUE, HIGH-VALUES, LOW-VALUE, LOW-VALUES, QUOTE, QUOTES, and ALL 'literal,' as defined within COBOL.

Hexadecimal literal

A string of 1-128 pairs of hexadecimal digits (0-F) bounded by pairs of quotes; the first quote must be preceded by a space or margin, and the last quote must be followed by a space, margin, or terminating punctuation character. Each digit pair represents one character, and the literal is valid in any input where a non-numeric literal is valid. The resulting non-numeric literal is used in translation.

Identifier

A data-name, unique in itself or made unique by the addition of qualifiers and/or subscripts or indices.

Keyword

A constant which can be defined within certain macro prototypes in addition to the macro-name. A keyword can be defined as literal. The compiler-directing words NOTE and REMARKS cannot be specified as keywords.

Label

A logical destination defined within a macro model. It takes the form of "&Lname".

Level indicator
The words FD, SD, RD, and CD that identify a specific type of file or a position in the data hierarchy.

Level number
A numeric literal in the DATA DIVISION immediately following a period or immediately following, consecutively, a period and an Area A indicator.

Literal
(See Figurative constant, Non-numeric literal, Hexadecimal literal, and Numeric literal)

Logical destination
Special words that precede directives, directive expressions, and substitution words in a macro model to declare a place within the model. If specified within a directive expression, the logical destination is a reference and names the location to which a transfer of control may be made.

Macro
The equation of a word, word pattern, or event in the input source program to pre-defined rules which generate output statements either through direct substitution or through conditional, logical analysis.

Macro call
The execution of a macro model. The execution can be initiated by a match of the macro prototype to words or events in the input stream or to a word substituted from another macro model.

Macro loading
The reading and assembling of macro models in preparation for translation.

Macro model
The specific translation action executed upon recognition of a source word, word prefix, syntax pattern, or event which matches the corresponding macro prototype.

Macro-name
The first word of a macro prototype.

Macro prototype
The explicit definition of the word, word pattern, or event to be compared to words and events in the input program. The sets can be independent or interdependent.

Macro set
A group of CA-MetaCOBOL+ macros designed to perform specific translation functions upon a COBOL or CA-MetaCOBOL+ input program. The sets can be independent or interdependent.

Model
(See Macro model)

Nest level
The level at which a macro definition containing an inner macro instruction is processed.

Non-numeric literal
A constant of 1-160 characters and spaces bounded by quotation marks. In CA-MetaCOBOL+, the quotation marks can be paired quotes (") or paired apostrophes(').

Numeric character
A character in the set 0-9.

Numeric literal
A constant containing from 1 to 18 digits. A numeric literal may contain a sign character as the left-most character and a decimal point. Floating point numeric literals, as defined within COBOL, are also classified as numeric literals.

Procedure header
One or more words in the PROCEDURE DIVISION preceded by an Area A indicator.

Procedure-name
A paragraph-name or section-name in the PROCEDURE DIVISION. When reference, it must be unique in itself or made unique by the addition of a section-name qualifier.

Prototype
(See Macro prototype)

Punctuation character
A period, comma, semicolon, or colon.

Qualifier
A data-name or section-name which is hierarchically superior to the data- or procedure-name being qualified. A qualifier must be separated from the name being qualified by the word IN or OF.

Region
The subdivision created between multiple macro sets by the use of the *$REGION Translator-directing statement. This regionalization permits identical global variables and label names to be defined in more than one macro set without causing conflict when they are fun in conjunction with each other.

Reserved variables
Several variables are predefined within the Translator and have specific uses. &VUPSI1-8, &VSOURCE, and &VDIALECT are predefined as external non-numeric variables. they cannot be specified within &EXTERN, &GLOBAL, or &LOCAL directives or as the receiving item in an &SET or &SETR directive. (Refer to the description of options in the *CA-MetaCOBOL+ User Guide*.)

Sign character
A plus (+) or minus (-) sign.

Space
One or more contiguous blanks (hexadecimal 40).

Special character

Any character other than an alphabetic or numeric character or a space.

Subscript

An acceptable subscript is a numeric literal, a character variable (as long as it has a numeric value), a numeric variable, and a reserved variable (as long as it has a numeric value).

Subscripts can be used anywhere a variable is allowed except in the &PIC and &SETTAB directives and line output.

Substitution word

A symbolic or constant word in a macro model which is substituted directly or indirectly in the generated output, and serves as word storage areas, as work areas during composition, as switches, etc.

Symbolic operand

The macro notation & or &n, where n is 0 through 15, used in writing macros which, during translation, represent the current value as determined by words from the input stream, by source item acquisition (&GET), or by equating (&EQU).

Tag

A logical destination defined within a macro model. It takes the form of &Tname.

TDS relative address

The serial number of every data-name entry in the TDS table (attribute table).

Terminating punctuation character

A punctuation character followed by a space, margin, or right parenthesis.

Translation

The modification, expansion, and analysis of standard COBOL or CA-MetaCOBOL+ input to produce COBOL output in accordance with the rules defined within the macro(s) invoked.

Translate-time

CA-MetaCOBOL+ Translator run time.

Variable

A storage area internal to macros.

Word

A string of characters fitting one of the following descriptions:

- Numeric literal.
- Non-numeric literal.
- Terminating punctuation character.
- Contiguous character string, not exceeding 30 characters, and bounded by a space, margin, or terminating punctuation character. The string may not contain an embedded space, quotation mark, or punctuation character. Such words can be a verb, reserved word, data-name, procedure-name, etc.

Index