# CA-MetaCOBOL™ +

## Structured Programming Guide

Release 1.1

R203M+11DRP

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

# Contents

# 1. About this Manual

## 1.1 Purpose

This manual is both a guide and reference for using the CA-MetaCOBOL+ Structured Programming (SP) Facility.

## 1.2 Organization

| Chapter | Description |
|---|---|
| 1 | Discusses the purpose of the manual, gives a list of CA-MetaCOBOL+ documentation, and explains notation conventions for CA-MetaCOBOL+. |
| 2 | Introduces the CA-MetaCOBOL+ Structured Programming (SP) Facility, including the support of the structured programming macro sets. |
| 3 | Describes how to define modules and invoke subordinate modules to create single entry, single exit modules. |
| 4 | Describes structured programming sequence, selection, and repetition constructs implemented in CA-MetaCOBOL+. |
| 5 | Discusses local data definition and unique control variables. |
| 6 | Discusses macro sets of the SP Facility. |
| 7 | Compares generated COBOL to structured source code. |
| 8 | Describes ways to access CA LIBRARIAN, manipulate bits, and sort tables. |
| Appendix A | Provides and describes a sample CA-MetaCOBOL+ SP program. |

| | |
|---|---|
| Appendix B | Describes various categories of generated data names and procedure names. |
| Appendix C | Contains the diagnostics for the Structured Programming Facility. |

## 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 Assuranc 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 VSI 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 Tutorial | Introduces the Macro Facility. Includes informatio on writing basic macros, model programming, macro writing techniques, and debugging. |
| Macro Facility Reference | Includes detailed information on the program flow c the CA-MetaCOBOL+ macro translator, macro format, definition of comments, macro nesting, macro prototypes, symbolic words, and model programming. |
| Quality Assurance Guide | Introduces the Quality Assurance Facility. Include 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. |

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 | 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*. |
| <u>Underscores</u> | 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

Minor technical and editorial revisions have been made throughout the manual.

# 2. Introduction

This chapter introduces structured programming with the Structured Programming (SP) Facility. It also disusses the benefits of using the SP Facility over IBM's VS COBOL II and standard COBOL.

## 2.1 What is Structured Programming?

Structured programming is a discipline that helps programmers develop programs efficiently. A structured program consists of single-entry, single-exit modules, uses selection and repetition control structures, and is consistently formatted and documented.

In a program, each module performs a single and distinct function. Together, the modules of a program form a single hierarchy: the top-most modules perform high-level tasks and the bottom-most modules perform low-level tasks.

The modular approach has the following advantages:

- Modules can be implemented and tested separately.
- Because modules form a single hierarchy, you can readily determine a module's role relative to other modules in the program. The hierarchy also identifies the order in which modules are developed: starting from the high-level modules and progressing to low-level modules. This technique is known as top-down development.
- A proper module has inputs and outputs that are easy to identify because a module may have only one entrance and one exit.

Properly structured programs contain statements taken from the sequence, selection, and repetition control structures. The SP Facility, as opposed to standard COBOL, provides statements that adequately implement the selection and repetition structures.

Consistent formatting and documentation of programs is an important task. It is easier to read programs that conform to standard formatting rules. More importantly, clear and complete documentation makes a program easier to maintain.

## 2.2 What is the SP Facility?

The SP Facility provides tools that support coding and debugging COBOL programs. For example, one component of the SP Facility is the COBOL language extensions that facilitate structured programming. Other components format and document SP programs automatically. The SP Facility provides top-down development, statements that fully support the selection and repetition constructs, and facilities for formatting and documenting programs.

SP programs are translated into standard COBOL by CA-MetaCOBOL+ and then compiled by your COBOL compiler. CA-MetaCOBOL+ and the SP Facility are compatible with all IBM COBOL compilers, including VS COBOL II.

The SP Facility enhances the COBOL language by providing true structured programming. The following outlines the structured programming services of the SP Facility.

**Facilitates Top-Down Development**

The SP Facility aids top-down development in a number of ways. It has statements that invoke subordinate modules and return control to the invoking module directly. It has a FLAG statement that enables explicit definition of control variables to true and false.

The SP Facility permits data to be defined locally within a module. This feature lets you define data in the Procedure Division. Then data referenced by a module can be defined within the module itself, making the program more readable.

The SP Facility also provides a facility for generating stubs (dummy modules) for uncoded modules that are referenced in a program. This feature lets programmers practice top-down development by making it possible to compile a program, even when a module invokes a subordinate module that has yet to be coded.

The statements for defining flags and local data are described in Chapter 5. Defining and invoking modules is discussed in Chapter 3.

**True Selection and Repetition Constructs**

The SP Facility selection and repetition constructs are true implementations of the structured programming techniques. The statements for coding these constructs enable programmers to develop programs with clear and concise logic.

The following are examples of SP Facility constructs for selection and repetition. (These examples do not represent complete syntax; refer to Chapter 4 for detailed syntax descriptions.)

**SP Facility Selection Constructs:**

```
IF condition
    process
ELSE
    process
ENDIF
```

```
        { FIRST   }
SELECT  { EVERY   }  ACTION
        { LEADING }

WHEN condition-1
   process-1
WHEN condition-2
  [process-2]

      { NONE    }
      { ANY     }
WHEN  { SOME    }  ARE SELECTED
      { NOT ALL }
      { ALL     }

ENDSELECT
```

```
SEARCH [ALL] ident
   WHEN END
      process-1
   WHEN condition
      process-2
ENDSEARCH
```

**SP Facility Repetition Construct:**

```
LOOP
    process
{ LEADING WHEN }
{ WHILE        }  condition
{ UNTIL        }

    process
ENDLOOP
```

**SP Facility Repetition Construct:**

```
LOOP
    process
{ LEAVE WHEN }
{ WHILE      }  condition
{ UNTIL      }

    process
ENDLOOP
```

A process can be one or more COBOL imperative statements or SP Facility constructs, or both. These constructs prohibit the use of GO TO statements or PERFORM...THRU statements. The scope terminators ENDIF, ENDSELECT, ENDLOOP, and ENDSEARCH promote unambiguous nesting. SP Facility statements are discussed in Chapter 4.

As previously shown, the SP Facility SELECT and SEARCH constructs provide a number of options for selection or case processing. For example, the SELECT construct provides optional NONE, ANY, SOME, ALL, and NOT ALL postscripts which are processed after the conditions have been tested. The LOOP construct lets you test the condition before and after processing takes place.

**Facilities for Documenting and Formatting Programs**

The SP Facility Structured Programming Documentor (SPD) and the High-Level Formatter (HLF) respectively document and format SP programs, providing a consistent level of program documentation and formatting. SPD creates a module hierarchy report showing the hierarchy of modules and a cross-referenced listing of where the modules are invoked. HLF indents, continues, and breaks lines of COBOL and SP code according to programmer or site defined formatting rules.

SPD, HLF, and other SP Facility macro sets are discussed in Chapter 6.

## 2.3 The SP Facility versus Conventional COBOL and VS COBOL II

The following table lists the characteristics of structured programming and indicates which types of COBOL support each of them.

| Feature of Structured Programming | Type of Support | CA's SP Facility | ANSI 68 or 74 COBOL | VS COBOL II |
|---|---|---|---|---|
| Enforces Adherence to Structured Programming | Mandatory single-entry single-exit modules | YES | NO | NO |
| | Scope terminators required | YES | NO | NO |
| | Provides for formatting of programs | YES | NO | NO |
| Top-Down Development | Explicit control variable definition | YES | NO | NO |
| | "Local" data definition in Procedure Division | YES | NO | NO |
| | Stub Generation | YES | NO | NO |
| | Documentation of module hierarchy and invocation | YES | NO | NO |
| Repetition Structure | DO-WHILE | YES | NO | YES |
| | DO-UNTIL | YES | NO | YES |
| | Process before and after test | YES | NO | NO |
| Selection (case) structure | SELECT FIRST | YES | NO | YES |
| | SELECT EVERY | YES | NO | NO |
| | SELECT LEADING | YES | NO | NO |

Conventional COBOL (1974 ANSI COBOL) does not support structured programming at all. While VS COBOL II (1985 ANSI COBOL) does support two variants of the repetition construct (PERFORM...UNTIL) and a limited form of the selection construct (EVALUATE), it does not approach the level of support provided by the SP Facility.

VS COBOL II has other limitations. It does not have facilities for supporting top-down module development or program formatting and documentation, both of which the SP Facility provides. For example, in VS COBOL II programs, you cannot define data locally within a module, the definition of control variables is not explicit, and there is no stub generation facility.

VS COBOL II permits noncompliance with the structured techniques. It does not enforce the use of scope terminators, which promote unambiguous nesting. Nor does it enforce single-entry single-exit modules because programmers can still use the GO TO and PERFORM…THRU statements. This lack of enforcement means that programmers can undermine structured programming standards. The SP Facility, on the other hand, promotes strict adherence to structured programming.

As described in Section 2.2, the SP Facility provides enhanced versions of the selection and repetition constructs. For example, the SELECT construct provides optional postscript clauses (WHEN NONE, WHEN SOME, and so on) that are processed after the primary clause(s) of the statement. The postcripts specify additional action depending on the number of true conditions found in the primary WHEN clauses. In addition, the SP Facility LOOP construct can test the condition before or after action is taken. (See Chapter 4 for details.)

## 2.4 Getting Started

This section describes the SP Facility macro sets and how to prepare JCL for running CA-MetaCOBOL+ for MVS and VSE systems.

CA-MetaCOBOL+ translates SP programs into standard COBOL. The output source program is then compiled by a standard IBM COBOL compiler. Input to CA-MetaCOBOL+ consists of translate-time options that modify CA-MetaCOBOL+ translation, the macro set desired for a translation, and the SP source program. Input may be provided directly from the Primary Input File or from source libraries, CA LIBRARIAN, or CA PANVALET.

Refer to the CA-MetaCOBOL+ *User Guide* for a complete list of CA-MetaCOBOL+ translate-time options and instructions on how to specify them.

### 2.4.1 SP Facility Macro Sets

Macro sets support various services of the SP Facility and are specified when CA-MetaCOBOL+ is executed. The following macro sets are provided with the SP Facility option of CA-MetaCOBOL+.

**GPV** General Programming Verbs. These verbs consist of statements for accessing CA LIBRARIAN and for performing bit manipulation and table sorting.

**HLF** High-Level Formatter. HLF formats the Procedure Division to provide consistent indentation and improve the readability of programs containing SP Facility statements, command-level CICS statements, and high-level statements provided by the DB and IMS Facilities.

**SPD** SP Documentor. SPD provides a module hierarchy and a module cross-referenced report. This macro set is not used with the SP2 macro set. SPD can be used with the SPP or HLF macro sets.

**SPP** This macro set provides the SP Facility syntax when the target compiler is ANSI 68, ANSI 74, or VS COBOL II.

**SP2** This macro set provides the SP Facility syntax when the target compiler is VS COBOL II. SP2 generates an output program in COBOL II structured syntax. SP2 also includes its own internal version of the SPD and SPS macro sets. These internal macro sets are enabled or disabled using UPSI Translate-time options. See Chapter 6 for details.

**SPS** SP Stub Generator. SPS defines modules that are referenced but have yet to be coded. This allows modules that reference undefined subordinate modules to be translated and compiled. This macro set is not used with the SP2 macro set. SPS can be used with the SPP macro set.

**Note**: Some macro sets cannot be loaded with other macro sets. See Chapter 6 for details.

### 2.4.2 Sample JCL for Executing CA-MetaCOBOL+

The following are brief examples of translation JCL for MVS and VSE systems, respectively. Refer to the CA-MetaCOBOL+ *User Guide* for more information.

In the examples, the Structured Programming Processor (SPP) macro set is loaded with either the CA-MetaCOBOL+ *$COPY or *$LIBED translator-directing statement. *$COPY loads macro sets from a standard source library; *$LIBED loads macro sets from a CA LIBRARIAN master file. Each statement must begin in column 7.

**Note:** The following JCL is sample start-up JCL only. You must supply the *options*, and *source program* indicated in 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.

The following JCL is supplied as a guide for executing CA-MetaCOBOL+ in an MVS environment. The SYSOUT class and BLKSIZE (in multiples of sizes shown), SPACE, and UNIT parameters may be altered.

```
 //[stepname] EXEC PGM=usermct[,PARM='options']
 //STEPLIB DD DSN=CA-MetaCOBOL+.loadlib,DISP=SHR
[//        DD DSN=LIBRARIAN.loadlib,DISP=SHR]      CA-MetaCOBOL+ load library
 //LSTIN   DD SYSOUT=A[,DCB=BLKSIZE=121]           Listing File
 //PUNCHF  DD SYSOUT=B[,DCB=BLKSIZE=80] COBOL Source Output File
 //AUX     DD SYSOUT=B[,DCB=BLKSIZE=80] Auxiliary File
 //SYSTERM DD SYSOUT=A[,DCB=BLKSIZE=80]            Terminal File
 //FE      DD UNIT=SYSDA,
              SPACE=(TRK,(20,10))[,DCB=BLKSIZE=148]Work File
 //FM      DD UNIT=SYSDA,
 //           SPACE=(TRK,(20,10))[,DCB=BLKSIZE=148]Work File

{//SYSLIB DD DSN=user.srclib,DISP=SHR   }          Source Library Input
{//MASTER DD DSN=user.disklibr,DISP=SHR }          CA LIBRARIAN input

 //CARDF DD *[,DCB=BLKSIZE=80]                Primary Input File
[OPTION options]

     {*$COPY SPP }                                 from standard library
     {*$LIBED SPP}                                 from CA LIBRARIAN master

 source program
 /*
```

**Note:** The following JCL is sample start-up JCL only. You must supply the *options*, and *source program* indicated in 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.

The job stream outlined below is supplied as a guide for executing CA-MetaCOBOL+ in a VSE environment.

```
 // LIBDEF ...
 // DLBL   IJSYS01,...
 // EXTENT SYS001,...                     Work File
 // ASSGN  SYS001,DISK,VOL=volserno,SHR
 // DLBL   IJSYS02,...
 // EXTENT SYS002,...                     Work File
 // ASSGN  SYS002,DISK,VOL=volserno,SHR
 // DLBL   IJSYS05,...                    CA LIBRARIAN input
 // EXTENT SYS005,...
 // ASSGN  SYS006,DISK,VOL=volserno,SHR
 // DLBL   IJSYS06,...                    To 'punch' CA-MetaCOBOL+
 // EXTENT SYS006,...                     output to a disk
 // ASSGN  SYS006,DISK,VOL=volserno,SHR
 // DLBL   IJSYS07,...
 // EXTENT SYS007                         Auxiliary output
 // ASSGN  SYS007,X'cuu'
 // DLBL   IJSYSSL,...                    Private source
 // EXTENT SYSSLB,...                     statement library
 // ASSGN  SYSSLB,...
 // EXEC ZMCTA
[OPTION options]

     { *$COPY SPP  }                     from standard library
     { *$LIBED SPP }                     from CA LIBRARIAN master

 source program
 /*
```

# 3. Module Definition and Invocation

The principles of module definition and invocation of subordinate modules are considered the key to top-down program development. Top-down program development concepts encourage both the designer and programmer to concentrate initially on the control module. The details of subordinate modules are deferred to a later time. This is accomplished by creating an abstraction, by a descriptive name, for various processes that the designer and programmer cannot consider in detail at the earliest stages of development. When programmed, the desired processes can be invoked by referring to their names. As program development continues, progressively lower level abstractions are designed and coded, until no further abstractions are needed.

This is the principle of stepwise modular refinement, and is based on the recognition that no person can effectively keep track of all details of a complex problem at once. Complex tasks must be broken down into manageable segments in an orderly fashion. High level modules control the program logic and contain little or no detailed processing logic. They act as an index to subordinate details for the developer, the reader, and the maintenance programmer. Low level modules contain few control mechanisms. Rather, they contain all the minor details of processing.

The Structured Programming Processor (SPP) provides a standard method for module definition and invocation.

## 3.1 Module Definition

Within a structured source program, a module is similar to a COBOL paragraph. It consists of a *module-name* and all the processes and control structures which follow until the next module-name or the end of the source program.

- Each module must have a module-name that begins in area A (columns 8-11). The module-name must be unique within the first 19 characters, must not exceed 30 characters in length, and must end with a period.
- No module can have an EXIT suffix. Do not create module names of the form *module-name*-EXIT.
- Each module has a single entry and a single exit. The entry immediately follows the module-name. The exit follows the final process or the end of the final control structure, whichever, is last.
- The first module of the Procedure Division, and any module containing an ENTRY statement, must end with either a STOP RUN or GOBACK statement.
- COBOL sections are permitted in the Procedure Division. However, a module-name must immediately follow the section name, and the exit point of this initial module becomes the exit point for the section.
- Within Declaratives, the name of the initial module of a section must follow the USE statement.

**Note:** The Translate-time option, UPSI5=G, relaxes the requirements that the initial module of the Procedure Division and all sections have module-names. The requirement that the initial module of the Procedure Division and any module containing an ENTRY statement end with a STOP RUN or GOBACK statement is relaxed.

**Caution**: Use of the Translate-time option, UPSI5=G, permits the initial module of the Procedure Division to fall through to a subsequent module.

## 3.2 Module Invocation

The process of invocation provides the control necessary for each module within a program to do its designated task and then return control to the invoking module.

Realizing the importance of invocation within structured programs, SPP provides a standard method for module invocation through the use of the simple PERFORM or DO verb.

The PERFORM or DO verb invokes the designated module-name:

## Format:

```
{ PERFORM }
{ DO      }  module-name
```

**module-name**

is a word that must comply with the COBOL requirements for defining a procedure-name: it must be unique within the first 19 characters; it must not exceed 30 characters in length; it must not end with a period; it cannot be formed as *module-name*-EXIT.

**Examples:** Each name in the example below refers to a specific module within the program and not to some local procedure-name. Also, each step within the PROCESS-INVOICE module is fully controlled.

There is a guarantee that when PERFORM POSSIBLE-CONTROL-BREAK completes its task (including any subordinate tasks), control returns to the next step, PERFORM AGE-INVOICE.

```
. . .
PROCESS-INVOICE.
    PERFORM POSSIBLE-CONTROL-BREAK
    PERFORM AGE-INVOICE
    PERFORM REPORT-AGED-INVOICE
. . .
```

The following DO verb example replicates the PERFORM verb example. The same conditions apply to each example:

```
. . .
PROCESS-INVOICE.
    DO POSSIBLE-CONTROL-BREAK
    DO AGE-INVOICE
    DO REPORT-AGED-INVOICE
. . .
```

**Notes:** PERFORM...THRU and SORT/MERGE...THRU cannot be specified, since SPP generates local procedure-names and includes the appropriate THRU clause.

Also not permitted are the PERFORM...TIMES and PERFORM...UNTIL forms, which have the undesirable attributes of both invocation and loop control that separate the loop from the controlling mechanism. In a program designed using the principles of stepwise modular refinement, this separation of function from control results in fragmentation of the refined design. Therefore, when these forms of the PERFORM are allowed, the modular implementation of the program is dictated by both the design and the requirements of the language.

SPP supports in-line control of equivalent repetition functions, permitting modular implementation of a program to be dictated solely by the design and not by the requirements of the COBOL language.

A program can become a subordinate module of a separately compiled program, and can be invoked by a CALL to the subordinate program's ENTRY point.

## 3.3 Use of the COBOL ENTRY Statement

An ENTRY statement in an SP program is treated as a self-contained paragraph. When CA-MetaCOBOL+ encounters an ENTRY statement, it ends the paragraph containing or preceding the ENTRY statement and starts a new paragraph. If a paragraph name does not immediately follow an ENTRY statement, CA-MetaCOBOL+ generates one.

# 4. Intra-module Control Structures

Structured programming control structures are those elements within a module that control the sequence of logic for selection and repetition constructs. These constructs grew out of the basic structured programming principle that all proper programs have but one entry point and one exit point. This principle is based on the experience that to permit multiple entries and exits is to invite programs that are difficult to read, verify, and maintain. To ensure a proper program, therefore, each component of the program must also have but one entry and one exit. Each module must be proper, and each element of code within a module must also be proper.

The elements of code within each module are the structures of structured programming. Only the following structures are permitted:

- A Sequence of one or more operations. In COBOL, the imperative statements, such as MOVE and ADD, are sequence elements. PERFORM and CALL, as used for subordinate module invocation, are also sequence elements.

- Selection of one or more sequence elements. In COBOL, the IF and other special conditionals are selection elements. Under structured programming concepts, a selection element has but a single entry and a single exit, and is, therefore, a sequence element at a higher level.

- Repetition of one or more sequence elements. In COBOL, iterative loops are controlled by the GO TO or by invoking PERFORM...TIMES or PERFORM...UNTIL statements. In structured programming terms, a repetition element also contains a single entry and a single exit, and is also, therefore, a sequence element at a higher level.

Since each of the elements described above can be considered a sequence element, they can become the building blocks used to define a module, just as a hierarchy of modules builds a program and a hierarchy of programs builds a system.

## 4.1 GO TO Alternative

The COBOL language does not support the discrete control structures, or constructs, that are required to transform these elements into clear, readable source code.

Many who have implemented structured programming standards in COBOL have restricted the use of GO TO because it can be used to violate the proper program principle. However, in a top-down, modular programming environment, elimination of the GO TO from COBOL also eliminates much of the control over loops and selection within a module. In COBOL, for example, local paragraph-names and GO TO statements are required to define the head and exit of a loop, to avoid deep nesting of conditionals, or to bypass certain instructions.

When restrictions upon the use of the GO TO are applied, even within small, or local segments of code, the COBOL programmer must turn to the PERFORM to control repetition and complex conditional logic. The PERFORM, however, requires the definition of subroutines, often in remote locations of the program. These subroutines are often extra modules. They are "extra" in that they exist in addition to the modules defined in the program design. They are required only by the lack of adequate selection and repetition constructs in the COBOL language.

SP Facility structured programming constructs represent the consistent, logical alternative to both the GO TO and superfluous modularization. The constructs define the delimiters of loops and the selection of logic paths, without the need for GO TO, local paragraph-names, and additional, arbitrary modules. They enable the program to clearly and consistently parallel the design, and thereby make the program easier to read, verify, and modify.

## 4.2 SP Facility Control Structures

The following sections define the SP Facility control structures for repetition and selection that are available with SPP. Also described are modifications to the COBOL SEARCH verb (a selection element) and the special COBOL conditionals (AT END, INVALID KEY, ON SIZE) for compatibility with the structured forms.

In the definition of control structure format, the familiar COBOL term imperative statement is replaced by the term process, indicating a broader interpretation than that permitted by standard COBOL. Process is defined as one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

This definition of process means that SPP allows constructs to be nested within each other. The only restriction placed on this nesting is that the subordinate construct must be defined in its entirety within the boundaries of the process in the primary construct. A subordinate construct cannot overlap the primary structure.

**Note**: In most of the descriptions of control structures, the word process in the format is replaced by DO *module-name* in the examples, so that the process can be described in the module-name, simply to add clarity to the examples. Readers should not infer that subordinate imperatives and/or control structures are illegal or discouraged.

## 4.3 Selection

One of the strong points of structured programming is that the selection process is recognized as a separate and unique function within a program. This recognition is important because, as any programmer knows, decision testing can become extremely complex. Deep nesting of COBOL IFs or using GO TO... DEPENDING ON can, like local paragraph-names and indiscriminate use of the COBOL GO TO, entangle the programmer in confused logic.

SPP provides the means for a programmer to control nested IFs through the use of ENDIFs, and to control complex multiple choice tests through the SELECT statements. In addition, the multiple conditions associated with the COBOL SEARCH verb are modified to conform to the requirements of a construct.

### 4.3.1 The IF Construct

The SPP IF construct is similar to the COBOL IF statement except that it is terminated by the keyword ENDIF.

#### Format:

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

**condition-1**
is any valid COBOL condition, including compound conditionals using AND or OR.

**THEN**
An optional keyword identifying the true path process as a result of the truth test in the condition.

**process-n**

One or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**ELSE**

A required keyword identifying the false path process as a result of the truth test in the condition.

**Example:** The following example requires the use of two ENDIFs, one to terminate each of the outstanding IF constructs. Note that by using ENDIF to terminate the inner IF construct, the ELSE side of the remaining non terminated IF includes DO WRITE-DETAIL.

This example poses a problem in COBOL: The terminating period, aside from being a rather inconspicuous terminator, requires that

- The nested condition must be placed in a subordinate PERFORMed module, or that
- The DO WRITE-DETAIL must follow both DO PROCESS-TYPE-A and DO PROCESS-TYPE-B.

```
. . .
    IF HEADER
        DO WRITE-HEADER
    ELSE
        IF DETAIL-TYPE-A
            DO PROCESS-TYPE-A
        ELSE
            DO PROCESS-TYPE-B
        ENDIF
        DO WRITE-DETAIL
    ENDIF
. . .
```

**Notes:** ENDIF is a required keyword identifying the physical and logical end of the IF construct.

The primary advantage of the structured IF...ELSE...ENDIF is realized in nested situations. ENDIF terminates only the most recent non terminated IF, whereas the COBOL IF and all subordinate nested IF's are terminated by a period.

If standardized COBOL conditionals, such as ON SIZE, AT END, and INVALID KEY, are to be used in SPP, they must be specified as special forms of SPP's IF...ELSE...ENDIF construct. See Section 4.6 for further information.

## 4.3.2 The SELECT Constructs

SPP supports four SELECT constructs for multi-path or cascading decisions. Each SELECT consists of four parts:

- A heading that specifies the nature of the test that follows.
- A body that contains a series of tests followed by the action to be taken if the test is true.
- Optional postscripts that specify additional actions depending on the number of successful tests.
- A keyword that identifies the physical and logical end of the construct.

The general format for each SELECT construct is as follows:

### Format:

```
 SELECT
 WHEN condition
    process
[
 WHEN postscript  ARE SELECTED
    process
]
 ENDSELECT
```

The WHEN *condition* clause identifies one or more conditions to be tested. After the last condition is evaluated, control passes to one or more *postscripts*, if specified. The postscripts specify action to be taken when:

**NONE**
no true conditions exist.

**ANY**
at least one true condition exists.

**ALL**
every condition is true.

**SOME**
at least one but not all true conditions exist.

**NOT ALL**
less than all true conditions exist.

In the chart given below, the relationship between the count of true conditions and postscripts is illustrated:

```
                    count of true
                    conditions

SELECT
WHEN . . .
WHEN . . .
WHEN . . .

                    0    1    2    3

WHEN NONE           T    F    F    F
WHEN ANY            F    T    T    T
WHEN SOME           F    T    T    F
WHEN NOT ALL        T    T    T    F
WHEN ALL            F    F    F    T

ENDSELECT
```

For example, if three conditions are tested and two are true, WHEN NONE is false, WHEN ANY is true, WHEN SOME is true, WHEN NOT ALL is true, and WHEN ALL is false.

The SOME, ALL, and NOT ALL postscripts are not available for use with the SELECT FIRST ACTION and SELECT FIRST ACTION FOR constructs; the following subsections separately describe the syntax for each SELECT construct.
The SELECT FIRST ACTION FOR Construct

The SPP SELECT FIRST ACTION FOR construct is equivalent to the classic structured case form in that it provides a multi-path decision capability based on the value of a single identifier.

## Format:

```
SELECT FIRST ACTION FOR identifier
WHEN  condition-1
      process-1
┌                        ┐
│WHEN  condition-2       │
│      process-2         │
└                        ┘
┌                        ┐
│WHEN { NONE } ARE SELECTED │
│     { ANY  }           │
│      process-3         │
└                        ┘
ENDSELECT
```

**identifier**
is any valid COBOL data-name.

**WHEN**
is a required keyword identifying a condition to be tested. Additional WHEN clauses can also be specified.

**condition-n**
is the most restricted of the SELECT constructs in that each condition must be defined as one or more literals or identifiers that are compared to the *identifier* for equality. Also, the only compound logical operator allowed is OR.

**process-n**
is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**WHEN NONE ARE SELECTED**
is an optional postscript specifying the action to be taken when no true conditions exist.

**WHEN ANY ARE SELECTED**
is an optional postscript specifying the action to be taken when at least one true condition exists.

**Example:** The following example illustrates how a specific identifier (TRANSACTION-CODE) can be compared against 4 possible conditions. As soon as the first true condition is met and the subordinate process is executed, control passes to the postscript.

```
. . .
    SELECT FIRST ACTION FOR TRANSACTION-CODE
    WHEN 'A'
        DO ADD-TRANSACTION
    WHEN 'C'
        DO CHANGE-TRANSACTION
    WHEN 'D'
        DO DELETE-TRANSACTION
    WHEN NONE ARE SELECTED
        DO ERROR-TRANSACTION
    ENDSELECT
. . .
```

**Notes**: SELECT and FOR are required keywords identifying the physical and logical start of the SELECT construct. FIRST and ACTION are optional keywords; if specified, neither can be omitted, i.e., one keyword cannot be specified without the other.

SELECT FIRST ACTION FOR is terminated by the keyword ENDSELECT, which is a required keyword identifying the physical and logical end of the SELECT construct.

As soon as the first true condition is met and the appropriate process is executed, control passes to the postscript.

With this form of SELECT, the postscript is limited to WHEN NONE and WHEN ANY. When NONE is specified, *process-3* is executed only if there are no true conditions. ANY means that *process-3* is executed when one condition is proven true. Note that there is the option of specifying only one postscript, neither postscript, or both postscripts.

SELECT FIRST ACTION FOR constructs can be nested to a maximum of nine levels.

## 4.3.3 The SELECT FIRST ACTION Construct

A more general version of the SELECT FIRST ACTION construct is shown below:

### Format:

```
SELECT FIRST ACTION
WHEN  condition-1
      process-1
[WHEN  condition-2                ]
[      process-2                  ]
[WHEN { NONE } ARE SELECTED       ]
[     { ANY  }                    ]
[      process-3                  ]
ENDSELECT
```

**WHEN**
is a required keyword identifying a condition to be tested. Additional WHEN clauses can also be specified.

**condition-n**
is any valid COBOL condition, including compound conditionals using AND or OR.

**process-n**
is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**WHEN NONE ARE SELECTED**
is an optional postscript specifying the action to be taken when no true conditions exist.

**WHEN ANY ARE SELECTED**
is an optional postscript specifying the action to be taken when at least one true condition exists.

**Example**: The following example illustrates that the order in which the conditions are specified is critical.

If AGE-OF-INVOICE is greater than 90, the subordinate module OVER-90 is invoked, and control is then passed to the postscript. If AGE-OF-INVOICE is less than or equal to 90, but greater than 60, then OVER-60 is invoked and control passes to the postscript. Invocation of CURRENT occurs only when AGE-OF-INVOICE is less than or equal to 30. If the WHEN clauses preceding the postscript were incorrectly reversed, the modules OVER-60 and OVER-90 would never be invoked.

```
. . .
    SELECT FIRST ACTION
    WHEN AGE-OF-INVOICE IS GREATER THAN 90
        DO OVER-90
    WHEN AGE-OF-INVOICE IS GREATER THAN 60
        DO OVER-60
    WHEN AGE-OF-INVOICE IS GREATER THAN 30
        DO OVER-30
    WHEN NONE ARE SELECTED
        DO CURRENT
    ENDSELECT
. . .
```

**Notes**: SELECT is a required keyword identifying the physical and logical start of the SELECT construct. FIRST and ACTION are optional keywords; if specified, neither can be omitted, i.e., one keyword cannot be specified without the other.

SELECT FIRST ACTION is terminated by the keyword ENDSELECT, which is a required keyword identifying the physical and logical end of the SELECT construct.

First, WHEN *condition-1* is evaluated; If it is true, *process-1* is executed and control passes to the postscript. If it is false, *condition-2* is evaluated. After the last condition is evaluated, control passes to the postscript. Since a single identifier does not have to be specified with this form of SELECT, the user has more freedom in defining the condition tests; however, the order specified is critical.

With this form of SELECT, the postscript is limited to WHEN NONE and WHEN ANY. When NONE is specified, *process-3* is executed only if there are no true conditions. ANY means that *process-3* is executed when one condition is proven true. There is the option of specifying only one, neither, or both postscripts.

The SELECT EVERY ACTION Construct

SPP's SELECT EVERY ACTION can be specified to handle a series of tests that continue regardless of success or failure:

## Format:

```
SELECT FIRST ACTION
WHEN  condition-1
      process-1
[WHEN  condition-2
      process-2                          ]
[          { NONE    }
           { ANY     }
 WHEN      { ALL     }  ARE SELECTED
           { SOME    }
           { NOT ALL }
      process-3                          ]

ENDSELECT
```

**WHEN**
: is a required keyword identifying a condition to be tested. Additional WHEN clauses can also be specified.

**condition-n**
: is any valid COBOL condition, including compound conditionals using AND or OR. The order in which the conditions are specified is at the discretion of the user.

**process-n**
: is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**WHEN NONE ARE SELECTED**
: is an optional postscript specifying the action to be taken when no true conditions exist.

**WHEN ANY ARE SELECTED**
: is an optional postscript specifying the action to be taken when at least one true condition exists.

**WHEN ALL ARE SELECTED**
: is an optional postscript specifying the action to be taken when every condition is true.

**WHEN SOME ARE SELECTED**
is an optional postscript specifying the action to be taken when at least one but not all true conditions exist.

**WHEN NOT ALL ARE SELECTED**
is an optional postscript specifying the action to be taken when less than all true conditions exist.

**Example:** SELECT EVERY ACTION is useful when multiple, interrelated tests are performed as part of a record validation procedure as illustrated below.

```
. . .
    SELECT EVERY ACTION
    WHEN NAME = SPACES
        DO NAME-ERROR
    WHEN ADDRESS = SPACES
        DO ADDRESS-ERROR
    WHEN ZIP = SPACES OR ZIP NOT NUMERIC
        DO ZIP-ERROR
    WHEN TERMS NOT NUMERIC
        DO TERMS-ERROR
    WHEN ANY ARE SELECTED
        DO ERROR-PROCESSING
    WHEN NONE ARE SELECTED
        DO VALID-PROCESSING
    ENDSELECT
. . .
```

**Notes:** SELECT and EVERY are required keywords identifying the physical and logical start of the SELECT construct. ACTION is an optional keyword.

SELECT EVERY ACTION is terminated by the keyword ENDSELECT, which is a required keyword identifying the physical and logical end of the SELECT construct.

This construct is used when selecting multiple true conditions. First, *condition-1* is evaluated. If it is true, *process-1* is executed and *condition-2* is evaluated. If *condition-1* is false, control passes to *condition-2* for evaluation. Only after the last condition is evaluated does control pass to the postscript.

With this form of SELECT, up to five postscripts can be specified. When NONE is specified, *process-3* is executed only if there are no true conditions. ANY means that *process-3* is executed when one condition is proven true. ALL means that *process-3* is executed only if all conditions are true. SOME means that *process-3* is executed when at least one but not all conditions are proven true. NOT ALL means that *process-3* is executed when less than all true conditions exist. One, a combination, all, or none of the postscripts can be specified.

Avoid the use of SELECT EVERY ACTION when using SPP to write programs executing under CICS (see Section 4.8).

The SELECT LEADING ACTIONS Construct

In SPP, SELECT LEADING ACTIONS terminates testing on the first failure. Cascading through the conditions continues only as long as the conditions prove true.

## Format:

```
SELECT FIRST ACTION
WHEN  condition-1
      process-1
┌                          ┐
│WHEN  condition-2         │
│      process-2           │
└                          ┘
┌                          ┐
│      ┌NONE   ┐           │
│      │ANY    │           │
│WHEN <│ALL    │> ARE SELECTED│
│      │SOME   │           │
│      └NOT ALL┘           │
└                          ┘
      process-3

 ENDSELECT
```

**WHEN**
: is a required keyword identifying a condition to be tested. Additional WHEN clauses can also be specified.

**condition-n**
: is any valid COBOL condition, including compound conditionals using AND or OR. The order in which the conditions are specified is critical.

**process-n**
: is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**WHEN NONE ARE SELECTED**
: is an optional postscript specifying the action to be taken when no true conditions exist.

**WHEN ANY ARE SELECTED**
: is an optional postscript specifying the action to be taken when at least one true condition exists.

**WHEN ALL ARE SELECTED**
: is an optional postscript specifying the action to be taken when every condition is true.

**WHEN SOME ARE SELECTED**
is an optional postscript specifying the action to be taken when at least one but not all true conditions exist.

**WHEN NOT ALL ARE SELECTED**
is an optional postscript specifying the action to be taken when less than all true conditions exist.

**Example:** The order in which the conditions are specified is always critical as illustrated in the following example of control break determination. Below MINOR-BREAK is invoked if there is a break in the minor and/or major control field.

```
. . .
    SELECT LEADING ACTIONS
    WHEN MAJOR-MINOR IS NOT EQUAL TO PREV-MAJOR-MINOR
        DO MINOR-BREAK
    WHEN MAJOR IS NOT EQUAL TO PREV-MAJOR
        DO MAJOR-BREAK
    WHEN ANY ARE SELECTED
        MOVE MAJOR-MINOR TO PREV-MAJOR-MINOR
        DO NEW-PAGE-HEADING
    ENDSELECT
. . .
```

**Notes:** SELECT and LEADING are required keywords identifying the physical and logical start of the SELECT construct. ACTIONS is an optional keyword.

SELECT LEADING ACTIONS is terminated by the keyword ENDSELECT, which is a required keyword identifying the physical and logical end of the SELECT construct.

First, *condition-1* is evaluated; if it is true, *process-1* is executed and *condition-2* is evaluated. This continues until either the last condition is evaluated or the first false condition is encountered. If the first or any subsequent condition proves false, control passes to the postscript, as when the last true condition is evaluated.

With this form of SELECT, up to five postscripts can be specified. When NONE is specified, *process-3* is executed only if there are no true conditions. ANY means that *process-3* is executed when one condition is proven true. ALL means that *process-3* is executed only if all conditions are true. SOME means that *process-3* is executed when at least one but not all conditions are proven true. NOT ALL means that *process-3* is executed when less than all true conditions exist. One, a combination, all, or none of the postscripts can be specified.

Avoid the use of SELECT LEADING ACTIONS when using SPP to write programs executing under CICS (see Section 4.8).

## 4.3.4 SEARCH Constructs

The COBOL SEARCH statement is a selection mechanism that is similar in structure to a SELECT construct. Like the COBOL IF, however, the COBOL SEARCH statement must be terminated by a period. In addition, only COBOL imperative statements may follow the various conditions. SPP relaxes these restrictions through two SEARCH formats. Each SEARCH construct consists of three parts:

- A heading that specifies the nature of the search.
- A body that contains a series of tests followed by the action to be taken if the test is true.
- A keyword that identifies the physical and logical end of the construct.

### Format:

```
SEARCH
[WHEN  END     ]
[      process ]
WHEN  condition
      process
ENDSEARCH
```

## The Basic SEARCH Construct

In order to treat the SEARCH statement as a construct and to permit subordinate processes following the conditionals, the standard COBOL SEARCH statement is modified by SPP.

### Format:

```
SEARCH  identifier-1 [VARYING { index-name-1 } ]
                              { identifier-2 }

[WHEN  END        ]
[      process-1  ]
 WHEN  condition-1
       process-2
[WHEN  condition-2]
[      process-3  ]
       ENDSEARCH
```

**VARYING**
is an optional clause specifying successive values.

**index-name-1**
is an index data item.

**identifier-n**
is any valid COBOL data-name.

**WHEN END**
is the optional WHEN END condition and is equivalent to the COBOL AT END condition. If specified, it must be the first WHEN clause.

**WHEN**
is a required keyword identifying a condition to be tested. Additional optional WHEN clauses can also be specified.

**condition-n**
is any valid COBOL condition, including compound conditionals using AND or OR.

**process-n**
is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**Example:** In the following example, assume that a small table is examined serially for a match to a search argument. The keys in the table are arranged in ascending sequence. The initialization of the index is accomplished by a SET statement preceding the SEARCH.

Notice that the third WHEN clause serves only to optimize the table search by providing a premature termination on the GREATER THAN condition.

```
. . .
FIND-TABLE-ENTRY.
    SET X1 TO 1
    SEARCH TABLE-ENTRY VARYING X1
    WHEN END
        DO TABLE-KEY-NOT-FOUND
    WHEN TABLE-KEY (X1) IS EQUAL TO ITEM-KEY
        DO TABLE-KEY-FOUND
    WHEN TABLE-KEY (X1) IS GREATER THAN ITEM-KEY
        DO TABLE-KEY-NOT-FOUND
    ENDSEARCH
. . .
```

**Notes:** SEARCH is a required keyword identifying the physical and logical start of the SEARCH construct.

SEARCH is terminated by the keyword ENDSEARCH, which is a required keyword identifying the physical and logical end of the SEARCH construct.

The requirements for *identifier-1*, *identifier-2*, *index-name-1*, and the VARYING clause are identical to those defined for the SEARCH ALL format of the COBOL SEARCH statement.

## The SEARCH ALL Construct

SPP also provides a binary search form of the SEARCH statement in the SEARCH ALL construct:

### Format:

```
SEARCH ALL  identifier-1
┌                  ┐
│WHEN  END         │
│      process-1   │
└                  ┘
WHEN  condition-1
      process-2

ENDSEARCH
```

**identifier-1**
is any valid COBOL data-name.

**WHEN END**
is the optional WHEN END condition and is equivalent to the COBOL AT END condition. If specified, it must be the first WHEN clause.

**WHEN**
is a required keyword identifying a condition to be tested.

**condition-n**
is a valid COBOL condition. Compound conditionals using AND are permitted; those using OR are not.

**process-n**
is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**Example:** The following example accomplishes the same task as the SEARCH...VARYING example, but uses the SEARCH ALL construct.

```
. . .
FIND-TABLE-ENTRY.
    SEARCH ALL TABLE-ENTRY
    WHEN END
        DO TABLE-KEY-NOT-FOUND
    WHEN TABLE-KEY (X1) IS EQUAL TO ITEM-KEY
        DO TABLE-KEY-FOUND
    ENDSEARCH
. . .
```

**Notes**: SEARCH and ALL are required keywords identifying the physical and logical start of the SEARCH construct.

SEARCH ALL is terminated by the keyword ENDSEARCH, which is a required keyword identifying the physical and logical end of the SEARCH construct.

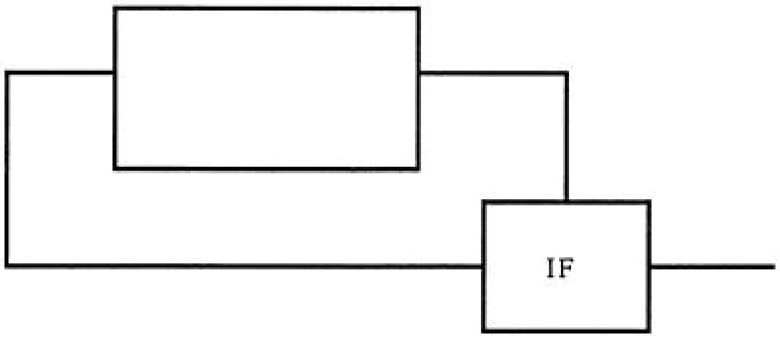
The requirements for *identifier-1* and *condition-1* are identical to those defined for the SEARCH ALL format of the COBOL SEARCH statement.

## 4.4 Repetition

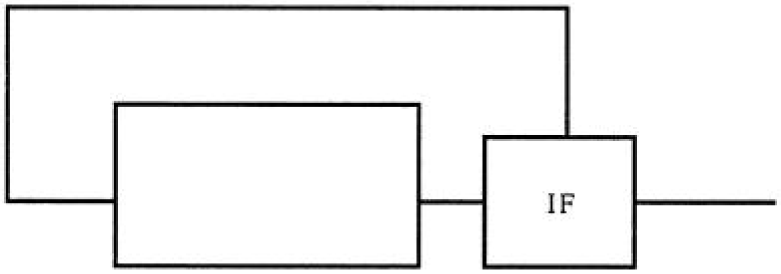
The disciplines of top-down program development and stepwise modular refinement require that repetition, or loops, be controlled and executed in-line.

In COBOL, the primary repetition functions are the PERFORM...TIMES and PERFORM...VARYING statements. These functions have the undesirable attributes of both invocation and loop control. Processing logic, therefore, is physically separated from the control mechanism.

The classic structured programming approach is to use the DO-WHILE and DO-UNTIL constructs for in-line repetition.

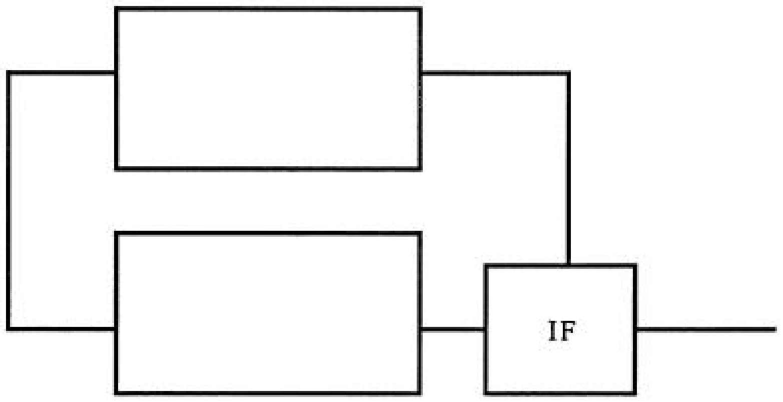


**DO-WHILE Construct**



**DO-UNTIL Construct**

The DO-WHILE tests for end of loop prior to the process, and the DO-UNTIL tests for end of loop following the process. However, in business data processing it is often necessary to include processes both before and after the condition test.. SPP resolves this by providing LOOP...ENDLOOP constructs.



**LOOP...ENDLOOP Construct**

SPP supports an in-line looping structure that uses the keywords LOOP and ENDLOOP to delimit the repetitions that are to occur. There are three formats of the LOOP construct. Each construct consists of three parts:

- A heading that specifies the nature of the loop.
- A body that specifies how long processing is to continue.
- A keyword that identifies the physical and logical end of the construct.

## Format:

```
LOOP
      process
{ LEAVE  WHEN condition }
{ WHILE  condition      }
{ UNTIL  condition      }
      process
ENDLOOP
```

## 4.4.1 The LOOP Construct

The first SPP LOOP format provides the basic structure necessary to handle the classic DO-WHILE and DO-UNTIL constructs:

### Format:

```
LOOP
       process-1
 ⎧                               ⎫
 ⎨ LEAVE  WHEN condition-1       ⎬
 ⎨ WHILE  condition-1            ⎬
 ⎩ UNTIL  condition-1            ⎭
       process-2
ENDLOOP
```

**LEAVE WHEN**
are keywords that specify processing continues until condition-1 is true.

**WHILE**
is a keyword that specifies processing continues as long as condition-1 is true.

**UNTIL**
is a keyword that specifies processing continues until *condition-1* is true.

**condition-1**
is any valid COBOL condition, including compound conditionals using AND or OR.

**process-n**
is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**Example:** The following example illustrates the use of WHILE. Processing continues as long as there are records left on the input file.

```
. . .
PROCEDURE DIVISION.
CONTROL-MODULE.
    DO BEGIN-PROGRAM
    LOOP
        DO READ-INPUT
    WHILE RECORDS-LEFT
        DO PROCESS-INPUT
    ENDLOOP
    DO END-PROGRAM
    STOP RUN
. . .
```

**Notes**: LOOP is a required keyword identifying the physical and logical start of the LOOP construct.

LOOP is terminated by the keyword ENDLOOP, which is a required keyword identifying the physical and logical end of the LOOP construct.

WHILE specifies that processing is to continue as long as *condition-1* is true. With LEAVE WHEN or UNTIL, processing continues until *condition-1* is true.

When *process-1* is omitted, the LOOP structure becomes the classic DO-WHILE construct testing for the end of the loop prior to the process;  when process-2 is omitted, the LOOP becomes the classic DO-UNTIL construct testing for the loop following the process.

## 4.4.2 The LOOP...TIMES Construct

The SPP LOOP...TIMES provides in-line repetition capabilities similar to those of the COBOL PERFORM...TIMES statement.

### Format:

```
LOOP { identifier } TIMES
     { integer    }

       process-1

{ LEAVE  WHEN condition-1 }
{ WHILE  condition-1      }
{ UNTIL  condition-1      }

       process
ENDLOOP
```

**identifier**
is a numeric data item with a positive integer value.  *Identifier* may contain the value 0, in which case the LOOP is not executed.

**integer**
is a numeric value.

**LEAVE WHEN**
are keywords that specify processing continues until *condition-1* is true.

**WHILE**
is a keyword that specifies processing continues as long as *condition-1* is true.

**UNTIL**
is a keyword that specifies processing continues until *condition-1* is true.

**condition-1**
is any valid COBOL condition, including compound conditionals using AND or OR.

**process-n**
is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**Example**: The following example shows the use of nested loops to reset a table element to spaces and zeros by class of item. Note that both LOOP structures must be terminated by their own ENDLOOP keywords. The first 10 occurrences of TABLE-ENTRY are reset, including the elementary numeric item TABLE-AMOUNT. Within each occurrence of TABLE-ENTRY, as shown by the inner LOOP, 5 occurrences of TABLE-UNITS are reset.

```
. . .
CLEAR-TABLE.
    SET X1 TO 1
    LOOP 10 TIMES
        MOVE SPACES TO TABLE-ENTRY (X1)
        MOVE ZEROS TO TABLE-AMOUNT (X1)
        SET X2 TO 1
        LOOP 5 TIMES
            MOVE ZEROS TO TABLE-UNITS (X1 X2)
            SET X2 UP BY 1
        ENDLOOP
        SET X1 UP BY 1
    ENDLOOP
. . .
```

**Notes:** LOOP and TIMES are required keywords identifying the physical and logical start of the LOOP construct.

LOOP...TIMES is terminated by the keyword ENDLOOP, which is a required keyword identifying the physical and logical end of the LOOP construct.

WHILE specifies that processing is to continue as long as *condition-1* is true. With LEAVE WHEN or UNTIL, processing continues until *condition-1* is true.

The loop terminates when *process-1* is executed the designated number of times or if LEAVE WHEN is specified when the condition is true. WHILE specifies that processing terminates when the condition is false. UNTIL specifies processing terminates when the condition is true.

## 4.4.3 The LOOP VARYING/FOR Construct

The SPP LOOP VARYING/FOR provides an in-line repetition alternative to the COBOL PERFORM...UNTIL statement.

### Format:

$$\text{LOOP}\ \left\{\begin{array}{l}\text{VARYING}\\ \text{FOR}\end{array}\right\}\ \left\{\begin{array}{l}\textit{index-name-1}\\ \textit{identifier-1}\end{array}\right\}$$

$$\left[\text{FROM}\ \left\{\begin{array}{l}\textit{index-name-2}\\ \textit{identifier-2}\\ \textit{integer-1}\end{array}\right\}\right]$$

$$\left[\text{BY}\ \left\{\begin{array}{l}\textit{identifier-3}\\ \textit{integer-1}\end{array}\right\}\right]$$

$$\left[\begin{array}{l}\text{UP THRU}\\ \text{DOWN THRU}\end{array}\right]\left\{\begin{array}{l}\textit{identifier-4}\\ \textit{integer-3}\end{array}\right\}$$

$$\textit{process-1}$$

$$\left\{\begin{array}{ll}\text{LEAVE} & \text{WHEN}\ \textit{condition-1}\\ \text{WHILE} & \textit{condition-1}\\ \text{UNTIL} & \textit{condition-1}\end{array}\right\}$$

$$\begin{array}{l}\textit{process-2}\\ \text{ENDLOOP}\end{array}$$

**VARYING**
Is a keyword specifying sequential repetition.

**FOR**
Is a synonym for VARYING.

**index-name-n**
Is an index data item.

**identifier-n**
Is a numeric data item with a positive integer value. If the *identifier* contains the value 0, the LOOP is not executed. When specified, *identifier-3* must be consistent with UP/DOWN (positive when UP is specified, negative when DOWN is specified).

**integer-n**
Is a numeric value. When specified, *integer-2* must be consistent with UP/DOWN (positive when UP is specified, negative when DOWN is specified).

**FROM**
Is a keyword specifying the initialization value. If the FROM clause is not specified, the default is 1.

**BY**
Is a keyword specifying the value of incrementation. If the BY clause is not specified, the default is 1.

**UP THRU**
**DOWN THRU**
Are keywords specifying the range of the incrementation or decrementation. If DOWN is not specified, the default is UP.

**process-n**
Is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**LEAVE WHEN**
Are keywords that specify processing continues until *condition-1* is true.

**WHILE**
Is a keyword that specifies processing continues as long as *condition-1* is true.

**UNTIL**
Is a keyword that specifies processing continues until *condition-1* is true.

**condition-1**
Is any valid COBOL condition, including compound conditionals using AND or OR.

**Example:** The following is an example of the LOOP VARYING construct. Note that it is a variation of the preceding LOOP...TIMES example.

```
. . .
CLEAR-TABLE.
    LOOP VARYING X1 FROM 1 BY 1 THRU 10
        MOVE SPACES TO TABLE-ENTRY (X1)
        MOVE ZEROS TO TABLE-AMOUNT (X1)
        LOOP VARYING X2 FROM 1 BY 1 THRU 5
            MOVE ZEROS TO TABLE-UNITS (X1 X2)
        ENDLOOP
    ENDLOOP
. . .
```

**Notes:** LOOP is a required keyword identifying the physical and logical start of the LOOP construct.

LOOP VARYING/FOR is terminated by the keyword ENDLOOP, which is a required keyword identifying the physical and logical end of the LOOP construct.

The primary termination of the LOOP occurs when the value of *index-name-1* or *identifier-1* is greater than (UP option) or less than (DOWN option) *identifier-4* or *integer-3*.

The loop terminates if the LEAVE WHEN option is specified when the condition is true. If the WHILE option is specified, process terminates when the condition is false. If the UNTIL option is specified, process terminates when the condition is true.

### 4.4.4 ESCAPE

In SPP, any LOOP...ENDLOOP construct can be terminated unconditionally by ESCAPE:

#### Format:

```
 ESCAPE
```

**Example:** The following example illustrates the use of ESCAPE:

```
. . .
LOOP
    PERFORM UPDATE-BUDGET
    IF WORK-TOTAL NEGATIVE
        ESCAPE
    ENDIF
ENDLOOP
. . .
```

**Notes:** When executed within a LOOP...ENDLOOP construct, ESCAPE causes the innermost LOOP in which it occurs to terminate.

Although the function of ESCAPE could be handled by a LEAVE WHEN, UNTIL, or WHILE conditional, ESCAPE implies an abnormal termination of the LOOP. Any reader of the program is alerted by the word ESCAPE that an exception condition has been encountered.

## 4.5 The Emergency EXIT

Leaving deep levels of a program in case of error or an unusual condition has always presented a problem. Top-down, modular, structured programming is no exception. The traditional approach has been to GO from the unusual condition TO a safe point within the program. Unfortunately, this approach invariably violates the single entry, single exit proper program principle.

SPP's EXIT can be specified within any process to provide a controlled way to leave a module of a program when an error or unusual condition occurs.

### Format:

```
EXIT
```

**Example:** In the following example, module UPDATE-TABLE uses a SEARCH construct to update an existing table element or insert a new table element. If, however, an insertion is required and table space is exhausted, an abnormal termination is in order. The WHEN END clause is used to signal the condition and EXIT the module. Control is passed to the process following DO UPDATE-TABLE in the test of TABLE-FULL-SWITCH. If the condition is true, ESCAPE causes transfer of control to DO CLOSE-ALL-FILES, and the program is terminated.

ESCAPE terminates only the LOOP and permits orderly termination of the program. EXIT, if used in place of ESCAPE in the control module, would bypass the STOP RUN.

```
. . .
PROCEDURE DIVISION.
CONTROL-MODULE.
    DO OPEN-ALL-FILES
    LOOP
        DO READ-INPUT
    WHILE RECORDS-LEFT
        DO LIST-INPUT-RECORD
        DO UPDATE-TABLE
        IF TABLE-FULL-SWITCH = 'YES'
            DO ABNORMAL-END-DISPLAY
            ESCAPE
        ELSE
            DO WRITE-OUTPUT
        ENDIF
    ENDLOOP
    DO CLOSE-ALL-FILES
    STOP RUN
```

```
. . .
UPDATE-TABLE.
    SET X1 TO 1
    SEARCH TABLE-ENTRY VARYING X1
    WHEN END
        MOVE 'YES' TO TABLE-FULL-SWITCH
        EXIT
    WHEN TABLE-KEY (X1) IS EQUAL TO ITEM-KEY
        DO UPDATE-TABLE-ENTRY
    WHEN TABLE-KEY (X1) IS GREATER THAN ITEM-KEY
    AND TABLE-KEY (LAST-ENTRY) = HIGH-VALUES
        DO INSERT-TABLE-ENTRY
     ENDSEARCH
. . .
```

**Notes:** The COBOL EXIT verb is not supported by SPP. The COBOL EXIT PROGRAM statement is an exception to the EXIT process. EXIT PROGRAM remains in the SPP-translated COBOL program, and is defined by COBOL.

EXIT should not be specified within the highest module in the top-down program hierarchy.

When an error or unusual condition occurs within a module, EXIT causes control to pass to the end of the module, which, in turn, passes control to the invoking module. Specifically, control returns to the process following the invoking process.

## 4.6 Standardized COBOL Conditions

If standard COBOL conditions, such as SIZE ERROR, END, OVERFLOW, END-OF-PAGE, DATA and INVALID, are to be used in SPP, they must be specified as special forms of SPP's IF...ENDIF construct (Section 4.3) as shown in the example below. They must also append COBOL verbs in the forms shown below.

```
COBOL       STANDARD COBOL CONDITIONS PLACED WITHIN SPP's
VERB       IF...ENDIF CONSTRUCT

⎧ADD       ⎫
⎪COMPUTE   ⎪
⎨DIVIDE    ⎬..    [IF [NOT]  ON SIZE ERROR process-1 [ELSE process-2] ENDIF]
⎪MULTIPLY  ⎪
⎩SUBTRACT  ⎭


VERB            IF...ENDIF CONSTRUCT

⎧DELETE  ⎫
⎨REWRITE ⎬ ...  [IF [NOT] INVALID KEY   process-1 [ELSE  process-2]  ENDIF]
⎩START   ⎭

RECEIVE   ...   [IF [NOT]   NO  DATA    process-1 [ELSE process-2] ENDIF]

READ      ...   [IF [NOT] ⎧INVALID KEY⎫ process-1 [ELSE process-2] ENDIF]
                          ⎩END        ⎭

                          ⎧END-OF-PAGE⎫
WRITE     ...   [IF [NOT] ⎨EOP        ⎬ process-1 [ELSE process-2] ENDIF]
                          ⎩INVALID KEY⎭

CALL      ...   [IF [NOT]  OVERFLOW     process-1 [ELSE process-2] ENDIF]

RETURN    ...   [IF [NOT]  AT END       process-1 [ELSE process-2] ENDIF]

STRING    ...   [IF [NOT]  OVERFLOW     process-1 [ELSE process-2] ENDIF]

UNSTRING  ...   [IF [NOT]  OVERFLOW     process-1 [ELSE process-2] ENDIF]
```

**. . .** (ellipsis)
an ellipsis that represents the remainder of the COBOL imperative/conditional statement.

**process-n**
is one or more COBOL imperative statements, one or more structured programming constructs, or any combination of COBOL imperatives and structured programming constructs. In addition, a process can be null, or omitted.

**Example:** The following example illustrates a READ…AT END condition test.

```
. . .
READ-MASTER.
    READ MASTER-FILE             [COBOL Conditional Statement]
    IF END
        CLOSE MASTER-FILE         SPP Standardized
    ELSE                          COBOL condition
        ADD +1 TO MASTER-FILE-COUNT
    ENDIF
```

## 4.7 The IS FALSE Condition Modifier

The IS FALSE condition modifier is an alternative to the COBOL NOT, and can be specified following a simple or compound conditional expression.

The IS FALSE condition modifier is permitted following a simple or compound conditional expression within an IF or SELECT construct with the following exceptions. IS FALSE is not permitted within a SELECT postscript, a SELECT FIRST ACTION FOR identifier, the SEARCH construct, the LOOP construct, or the standardized conditionals, such as IF END and IF SIZE ERROR.

For example, suppose that a condition-name, NO-MORE-INPUT, is defined to indicate end of file. It might then be convenient to test for the condition within a LOOP construct, as follows:

```
. . .
    LOOP
        DO READ-INPUT-RECORD
    LEAVE WHEN NO-MORE-INPUT
        DO PROCESS-INPUT-RECORD
    ENDLOOP
. . .
```

In the above example, the reference to NO-MORE-INPUT is natural and easily understood by a reader of the program. However, the choice of NO-MORE-INPUT as a condition-name may lead to a more difficult expression in another module that must test for MORE-INPUT, as follows:

```
. . .
    IF NOT NO-MORE-INPUT
. . .
```

The above conditional expression is somewhat confusing. Therefore, SPP permits the following as an equivalent form of negation:

```
. . .
    IF NO-MORE-INPUT IS FALSE
. . .
```

This technique is especially useful when handling complex editing operations, where it is often more clear to test for valid conditions, as follows:

```
. . .
    SELECT EVERY ACTION
    WHEN (TRANS-CODE = 'A' OR 'D') IS FALSE
        DO TRANS-CODE-ERROR
    WHEN (TRANS-AMOUNT > ZERO AND < 2000) IS FALSE
        DO TRANS-AMOUNT-ERROR
    WHEN ANY ARE SELECTED
        DO REJECT-TRANS-RECORD
    WHEN NONE ARE SELECTED
        DO UPDATE-MASTER-RECORD
    ENDSELECT
. . .
```

## 4.8 CICS Command Level Coding

Certain precautions should be taken when using SPP to write programs executing under CICS.

- Avoid the use of SELECT EVERY ACTION and SELECT LEADING ACTIONS control structures.

- The CA-MetaCOBOL+ translation must precede the CICS pre-compile.

These control structures generate a control variable in the Working-Storage Section of the program to support the use of postscripts. They should only be used when the value of the control variable can be guaranteed, or when the postscripts are not used.

**Example:** The following program illustrates a coding technique for HANDLE CONDITION. This option of the EXEC CICS control structure is translated into a GO TO... DEPENDING ON... statement, which transfers control to the indicated module. As long as these modules end with a CICS control structure which terminates the CICS session, or reference a module which ends the CICS session, top-down processing logic is preserved.

```
*$LIBED SPP
IDENTIFICATION DIVISION.
PROGRAM-ID. CICSXMPL.
 .  .  .
WORKING-STORAGE SECTION.
01  ABEND-CODE           PIC X(04).
 .  .  .
01  MESSAGE-RECORD.
 .  .  .
PROCEDURE DIVISION.
START-UP.
    MOVE LOW-VALUES TO MESSAGE-RECORD
    EXEC CICS
        HANDLE CONDITION
        NOTOPEN(FILE-NOT-OPEN)
        NOTFND(RECORD-NOT-FOUND)
        LENGERR(LENGTH-ERROR)
        IOERR(IO-ERROR)
        ILLOGIC(VSAM-ILLOGIC)
    END-EXEC
 .  .  .
FILE-NOT-OPEN.
    MOVE 'FILE NOT OPENED'  TO MESSAGE-RECORD
    MOVE 'VS01' TO ABEND-CODE
    DO DISPLAY-ERROR
RECORD-NOT-FOUND.
    MOVE 'RECORD NOT FOUND' TO MESSAGE-RECORD
    MOVE 'VS02' TO ABEND-CODE
    DO DISPLAY-ERROR
LENGTH-ERROR.
    MOVE 'LENGTH ERROR'     TO MESSAGE-RECORD
    MOVE 'VS04' TO ABEND-CODE
    DO DISPLAY-ERROR
IO-ERROR.
    MOVE 'IO ERROR'         TO MESSAGE RECORD
    MOVE 'VS05' TO ABEND-CODE
    DO DISPLAY-ERROR
VSAM-ILLOGIC.
    MOVE 'VSAM ILLOGIC'     TO MESSAGE RECORD
    MOVE 'VS06' TO ABEND-CODE
    DO DISPLAY-ERROR
DISPLAY-ERROR.
    EXEC CICS
        SEND TEXT
        FROM MESSAGE-RECORD
        LENGTH(106)
        ERASE
    END-EXEC
    EXEC CICS
        ABEND
        ABCODE(ABEND-CODE)
        CANCEL
    END-EXEC
```

# 5. Local Data Definitions and Control Variables

In adopting a modular approach to program implementation, two problems stand out:

- One is the monolithic nature of the Data Division which necessitates a search of all the entries in order to locate an item that is referenced by only one or two modules. The program would be far more readable if definitions of data structures that are *local* to a module were permitted within the module itself.

- The other problem deals with communications between invoking and invoked modules. This can be controlled by flags--two-valued control variables. By providing a consistent method for defining, testing, and modifying control variables, the reader is able to clearly distinguish between operations on data and operations on control variables.

This section defines the facilities of SPP to resolve both of these problems.

## 5.1 START DATA/END DATA and Local Data Definitions

SPP's START DATA allows data to be defined within the module which references it:

### Format:

START DATA

$$\left[\begin{array}{l}\textit{level-number-1 data-name-1}\\ \textit{level-number-2 data-name-2}\end{array}\right]$$

END DATA

**level-number-n**
is any valid COBOL level-number.

**data-name-n**
is any data item that is valid within the Working-Storage Section. Each data item defined must be terminated with a period.

**Example:** The following example shows the data definitions within the module to which they apply. Note that the items are defined before they are referenced.

```
. . .
PROCEDURE DIVISION.
. . .
POSSIBLE-CONTROL-BREAK.
    START DATA
    77  PREVIOUS-CUSTOMER  PIC X(7) VALUE LOW-VALUE.
    01  CUSTOMER-AREA.
        02 CUSTOMER        PIC X(7).
        02 INVOICE-NO      PIC X(7).
        02 INVOICE-DATE    PIC X(6).
        02 INVOICE-AMT     PIC S9(9)V99 COMP-3.
    END DATA
    IF CUSTOMER = PREVIOUS-CUSTOMER . . .
. . .
```

**Notes:** START DATA is terminated by the keyword END DATA. END DATA ends the free-standing data definition. START DATA allows the definition of free-standing data definitions. It must:

- Begin in Area B
- Immediately follow a module-name (a paragraph header)
- Define Working-Storage items only
- Precede any reference to the items defined

## 5.2 Control Variables

SPP provides for the precise definition of control variables. These definitions may be placed either in Working-Storage, if they are common to the program as a whole, or within a module, delimited by START DATA and END DATA (see Section 5.1), if they are referenced locally.

### 5.2.1 The FLAG Statement

SPP's FLAG defines, tests, and modifies control variables:

**Format:**

$$\textit{level-number}\ \text{FLAG}\ \textit{flag-name}\ \text{IS}\ \begin{bmatrix}\text{TRUE}\\ \text{FALSE}\end{bmatrix}$$

**level-number**
01-49, or 77.

**flag-name**
is a word that must comply with COBOL data-name requirements. It must be unique in the first 25 characters.

**IS TRUE**
**IS FALSE**
The variable is set to the TRUE or FALSE condition indicated.

**Example:** The following example illustrates the use of FLAG.

```
. . .
PROCEDURE DIVISION.
AGED-TRIAL.
    START DATA
    77  FLAG RECORDS-LEFT IS TRUE.
    END DATA
. . .
    LOOP
        DO GET-NEXT-INVOICE
    WHILE RECORDS-LEFT
        DO PROCESS-INVOICE
    ENDLOOP
. . .
```

**Note:** FLAG generates a 1-character data item with a default setting of FALSE, unless TRUE is specified.

## 5.2.2 The SET-TRUE/SET-FALSE Statement

The status of the control variable can be modified in the following format by SPP:

### Format:

$$\left\{\begin{array}{l}\text{SET-TRUE}\\\text{SET-FALSE}\end{array}\right\}\ \textit{flag-name-1}\ [\textit{flag-name-2}]$$

**flag-name-n**

is a word that must comply with COBOL data-name requirements. It must be unique in the first 25 characters.

**Example:** Using the previous illustration of FLAG, the control variable RECORDS-LEFT can be modified as follows.

```
. . .
GET-NEXT-INVOICE.
    START DATA
    77  FLAG RECORDS-LEFT IS TRUE.
    END DATA
    READ INVOICE-RECORD
    IF END
        SET-FALSE RECORDS-LEFT
    ENDIF
. . .
```

**Note:** With SET-TRUE/SET-FALSE, the variable is set to the TRUE or FALSE condition indicated.

# 6. Structured Programming Support Macros

In addition to the SPP and SP2 macro sets, the SP Facility also provides assistance in other areas of concern in structured programming. The following sections discuss these other macro sets and how they are used with SPP and SP2.

## 6.1 Structured Programming Documentor (SPD & SPDCOB)

The Structured Programming Documentor (SPD) provides additional documentation beyond the listing of the program. When SPD is appended to the Structured Programming Processor (SPP) or the High-Level Formatter (HLF), or when SPD is enabled by the UPSI4=X Translate-time option for SP2, SPD extracts raw data and places it on the Auxiliary Output File. Following translation, this raw data is used as input to the reporting program, SPDCOB. The reporting program generates two types of reports, the Module Hierarchy and the Module Where Invoked reports.

### 6.1.1 Reports Produced

This section illustrates the Module Hierarchy and Module Where Invoked reports that can be produced by the Structured Programming Documentor (SPD).

- Module Hierarchy--shows the hierarchical structure of a program (by level) in a format similar to that of a record description in the Data Division:

```
AGEDTB        M O D U L E   H I E R A R C H Y

01 AGED-TRIAL
   02  BEGIN-AGED-TRIAL
   02  GET-NEXT-INVOICE
   02  PROCESS-INVOICE
      03  POSSIBLE-CONTROL-BREAK
         04  CUSTOMER-BREAK
```

```
            03  AGED-INVOICE
            03  REPORT-AGED-INVOICE
                04  POSSIBLE-PAGE-BREAK
       02  END-AGED-TRIAL
         03  CUSTOMER-BREAK
         03  FINAL-BREAK
```

All modules invoked by a PERFORM (or DO) are reported, with the exception of Declaratives sections and paragraphs. Module-names invoked by CALL statements are prefixed on the report by **C=.

- Module Where Invoked--identifies each module in the program, the line number where each is defined, and the line number for each reference to a module.

```
AGEDTB       M O D U L E   W H E R E   I N V O K E D

DEFN                                REFERENCES

AGE-INVOICE                         001630   001450
AGED-TRIAL                          001190
BEGIN-AGED-TRIAL                    001320   001230
CUSTOMER-BREAK                      001920   001590   001490
END-AGED-TRIAL                      001480   001290
FINAL-BREAK                         002110   001500
GET-NEXT-INVOICE                    001370   001250
POSSIBLE-CONTROL-BREAK              001530   001440
POSSIBLE-PAGE-BREAK                 002210   001700
PROCESS-INVOICE                     001430   001270
REPORT-AGED-INVOICE                 001680   001460
```

Notice that the Module Where Invoked Report is sorted into sequence by module-name. Sections and paragraphs defined within Declaratives are neither reported nor CALLed module-names. By default, line numbers are reported as embedded sequence numbers (normally columns 1-6). You can optionally select to include CA-MetaCOBOL+ generated line numbers on the Input Listing.

### 6.1.2 Raw Data Extraction

When the Structured Programming Documentor (SPD) is used with the Structured Programming Processor (SPP) or the High-Level Formatter (HLF), SPD must be loaded in conjunction with and immediately following SPP or HLF, and must be within the same macro region.

Do not load SPD with SP2; SPD is built into SP2. The UPSI Translate-time option that controls SPD is:

- UPSI4=X enables the raw data extraction. This extraction is bypassed by default. X is initialized with a blank ccharacter to suppress use of SPD with SP2. To enable SPD, X must be replaced with any non-blank character.

During translation, SPD extracts information relating to each module identified within the program and writes the information to the Auxiliary Output File for subsequent input to the reporting program, SPDCOB. Extraction and reporting can be controlled, in part, by the following UPSI= Translate-time options:

- UPSI8=X suppresses the Module Hierarchy Report. This report is produced by default.

- UPSI7=X suppresses the Module Where Invoked Report. This report is produced by default.

- UPSI6=X selects the CA-MetaCOBOL+ generated line numbers on the Input listing in the Module Where Invoked Report. Line numbers from the embedded sequence number field, normally columns 1-6 of each source record are selected by default.

### 6.1.3 Generating Reports

SPDCOB is the source program which produces the documentation reports. SPDCOB is written in SPP syntax. It must, therefore, be translated by CA-MetaCOBOL+ under the control of SPP for compilation, link-edit, and execution. Refer to the CA-MetaCOBOL+ *VSE Installation Guide* or the CA-MetaCOBOL+ *MVS Installation Guide* for further information.

The JCL required to execute SPDCOB under MVS is shown below:

```
 //jobname   JOB ...
 //stepname EXEC PGM=SPDCOBname
[//STEPLIB    DD DSN=user.loadlib,DISP=SHR]
 //SYSOUT     DD SYSOUT=A
 //SYSPRINT   DD SYSOUT=A
 //SYSSORT    DD UNIT=SYSDA,SPACE=(TRK,20)
 //SORTWK01   DD UNIT=SYSDA,SPACE=(TRK,20)
 //SORTWK02   DD UNIT=SYSDA,SPACE=(TRK,20)
 //SORTWK03   DD UNIT=SYSDA,SPACE=(TRK,20)
 //SYSIN      DD DSN=dataset,DISP=(OLD,DELETE)
```

The JCL required to execute SPDCOB under VSE is shown below, where SPD-produced input is acquired from SYS010 and reports are written to SYS011:

```
// JOB ...
// ASSGN SYS010,X'cuu'
// ASSGN SYS011,X'cuu'
// ASSGN SYS001,X'cuu'
// ASSGN SYS002,X'cuu'
// ASSGN SYS003,X'cuu'
// DLBL SORTWK1,,68/001,SD
// EXTENT SYS001,xxxxxx,1,0,20,100
// DLBL SORTWK2,,68/001,SD
// EXTENT SYS002,xxxxxx,1,0,120,100
// DLBL SORTWK3,,68/001,SD
// EXTENT SYS003,xxxxxx,1,0,220,100
// EXEC SPDCOBname SPD-generated input
/*
/&
```

If you are using CA-MetaCOBOL+/PC, use the equivalent SET statements below.

```
SET SYSIN = filename[u]
SET SYSPRINT = filename[u]
SET SYSSORT = filename
SPDCOB
```

## 6.2 High-Level Formatter (HLF)

The High-Level Formatter formats the Procedure Division of a program to provide standard indentation and improved readability. The output of HLF is a formatted version of the original program.

HLF formats programs containing SP Facility, DB Facility, IMS Facility, and command-level CICS statements. Programs can contain any combination of these high-level languages. To format a program using the HLF macro set, specify the translate-time option:

### Format:

UPSII = $\begin{Bmatrix} \text{C} \\ \text{S} \end{Bmatrix}$

where *C* indicates that the program is to be formatted as a COBOL program, and *S* indicates that the program is to be formatted as an SP program.

**Example:** The following example shows an SP Facility statement after it was run through CA-MetaCOBOL+ with HLF. The FORMAT=PI4X2 translate-time option was used to specify two levels of indentation, with each level indented four columns.

**CA-MetaCOBOL+ Input**

```
. . .
module-name
    IF A
    MOVE U TO V
    ELSE
    IF B
    MOVE W TO X
    ELSE
    IF C
     MOVE Y TO Z
MOVE Y TO Z
     ENDIF
     ENDIF
     ENDIF
. . .
```

**CA-MetaCOBOL+ Output**

```
 . . .
module-name
    IF A
        MOVE U TO V
    ELSE
        IF B
            MOVE W TO X
        ELSE
                IF C



                        ENDIF
                    ENDIF
                ENDIF
         . . .
```

**Notes:** SPD is the only macro set that may be loaded with the HLF macro set in a single translation. If SPD is used in a translation, HLF must be loaded before SPD.

The number of columns indented and the number of levels of indentation are controlled by the FORMAT=(PIaXb) translate-time option, where *a* defines the number of columns indented within a level, and *b* defines the number of levels indented. Refer to Appendix C of the CA-MetaCOBOL+ *User Guide* for complete details concerning the FORMAT= translate-time option.

In SP programs, periods encountered in the Procedure Division that do not terminate a paragraph name, section name, or COPY statement are removed.

HLF formats programs containing statements for ANSI 68 and 74 target compilers, as well as statements for VS COBOL II target compilers.

## 6.3 Structured Programming Stub (SPS) Generator

The Structured Programming Stub (SPS) Generator can be invoked during an SPP or SP2 translation to automatically generate stubs for undefined modules.

When SPS is used with SPP, SPS must be loaded in conjunction with and immediately following the SPP macro set, within the same macro region.

Do not load SPS with SP2; SPS is built into SP2. The UPSI Translate-time option that controls SPS is:

- UPSI3=X enables the stub generation. Stub generation is bypassed by default. X is initialized with a blank character to suppress use of SPS with SP2. To enable SPS, X must be replaced with any non-blank character.

During top-down development, modules are tested as they are written. In order to ensure that a program is compilable throughout its development, dummy modules (stubs) may be needed. For example, a higher level module, during testing, may reference a procedure-name (module-name) which is not yet specified. Without a stub, the program is not compilable.

SPS automatically checks the program to determine if there are any references to undefined modules. If there are, each procedure-name invoked by the referencing modules is generated as a separate, empty module.

On the Input listing, any undefined modules are reported as illustrated below:

```
           . . .
          PROCEDURE DIVISION.
          MAIN-MODULE.
              DO MODULE-A
              DO MODULE-B
              DO MODULE-C
           . . .
          MODULE-A.
           . . .
          MODULE-C.
           . . .
     ***NOTE   FOLLOWING STUBS FOR UNRESOLVED DO'S
     ***NOTE        MODULE-B
```

SPS does not prohibit the programmer from defining a stub. For example, a programmer may want to write a dummy module containing special code--perhaps to return a sample of formatted data in order to test higher level modules.

SPS contains a null word-type macro named $STUB that is invoked at the time the stub is generated. Optionally, the user may override the $STUB macro on a list-in basis to include their own code in each generated stub. When $STUB is invoked, &1 contains the stub module-name within a non-numeric literal.

**SPP Example:** To produce a message containing the stub name whenever the stub is invoked, the following CA-MetaCOBOL+ input stream can be prepared:

```
*$COPY  SPP      /*  LOAD SPP MACRO, THEN --
*$COPY  SPS      /*  LOAD STUB GENERATOR,  THEN --
WP  $STUB :      /*  OVERRIDE $STUB MACRO.
DISPLAY &1.
(structured source program)
. . .
```

**SP2 Example:** Provided SPS has been enabled by way of the UPSI3=X Translate-time option, the following CA-MetaCOBOL+ input stream can be prepared to produce a message containing the stub name whenever the stub is invoked:

```
*$COPY  SP2      /*  LOAD SP2 MACRO, THEN --
WP  $STUB :      /*  OVERRIDE $STUB MACRO.
DISPLAY &1.
(structured source program)
. . .
```

# 7. Reading, Verifying, and Maintaining Structured Programs

The Structured Programming (SP) Facility structured programming macro sets provide a structured programming environment to enhance the COBOL language. The structured programming macro sets produce source code that is easy to read and understand. To achieve this higher level of program integrity and clarity, however, it is necessary to develop, verify, and maintain the structured programs at the CA-MetaCOBOL+ level.

## 7.1 A Comparison of Structured and COBOL Source

When the structured program is coded or when it reaches a level where testing can begin, the program is input to CA-MetaCOBOL+ for translation. The Translator, using the SPP macro set, generates compilable COBOL code. This results in the movement of local data descriptions, START DATA...END DATA, to the Working-Storage Section and the generation of local paragraph-names and GO TO statements in the Procedure Division. The translated program is then ready to be compiled, link-edited with any necessary support modules, and executed.

The distinction between the structured and COBOL levels becomes clearer by comparing the following examples. The first example shows a structured source module.

## Structured Source Module

```
001680 REPORT-AGED-INVOICE.
001690     ADD 1 TO NUMBER-OF-LINES
001700     PERFORM POSSIBLE-PAGE-BREAK
001710     MOVE SPACE TO AGED-LINE
001720     MOVE INVOICE-CUSTOMER TO AGED-LINE-CUSTOMER
001730     MOVE INVOICE-NO TO AGED-LINE-INVOICE
001740     MOVE INVOICE-DATE TO AGED-LINE-DATE
001750     SELECT FIRST ACTION
001760     WHEN AGE-OF-INVOICE IS GREATER THAN 90
001770         MOVE INVOICE-AMOUNT TO OVER-90-AMOUNT
001780         ADD INVOICE-AMOUNT TO CUSTOMER-OVER-90
001790     WHEN AGE-OF-INVOICE IS GREATER THAN 60
001800         MOVE INVOICE-AMOUNT TO OVER-60-AMOUNT
001810         ADD INVOICE-AMOUNT TO CUSTOMER-OVER-60
001820     WHEN AGE-OF-INVOICE IS GREATER THAN 30
001830         MOVE INVOICE-AMOUNT TO OVER-30-AMOUNT
001840         ADD INVOICE-AMOUNT TO CUSTOMER-OVER-30
001850     WHEN NONE ARE SELECTED
001860         MOVE INVOICE-AMOUNT TO CURRENT-AMOUNT
001870         ADD INVOICE-AMOUNT TO CUSTOMER-CURRENT
001880     ENDSELECT
001890     WRITE AGED-TRIAL-LINE FROM AGED-LINE
001900         AFTER ADVANCING 1 LINES
```

Notice the use of the SELECT...ENDSELECT construct which delineates the multiple-choice test. Compare the clarity of this code against the following example of the same module after expansion to conventional COBOL source input for compilation.

## COBOL Source Code

```
001680 REPORT-AGED-INVOICE.
001690     ADD 1 TO NUMBER-OF-LINES
001700     PERFORM POSSIBLE-PAGE-BREAK THRU POSSIBLE-PAGE-BREAK-EXIT
001710     MOVE SPACE TO AGED-LINE
001720     MOVE INVOICE-CUSTOMER TO AGED-LINE-CUSTOMER
001730     MOVE INVOICE-NO TO AGED-LINE-INVOICE
001740     MOVE INVOICE-DATE TO AGED-LINE-DATE
001760     IF AGE-OF-INVOICE GREATER THAN 90
001761         NEXT SENTENCE
001762     ELSE
001763         GO TO REPORT-AGED-INVOICE-SC-1-101.
001770     MOVE INVOICE-AMOUNT TO OVER-90-AMOUNT
001780     ADD INVOICE-AMOUNT TO CUSTOMER-OVER-90
001781     GO TO REPORT-AGED-INVOICE-SS-1.
001782 REPORT-AGED-INVOICE-SC-1-101.
001790     IF AGE-OF-INVOICE GREATER THAN 60
001791         NEXT SENTENCE
001792     ELSE
001793         GO TO REPORT-AGED-INVOICE-SC-1-102.
001800     MOVE INVOICE-AMOUNT TO OVER-60-AMOUNT
001810     ADD INVOICE-AMOUNT TO CUSTOMER-OVER-60
001811     GO TO REPORT-AGED-INVOICE-SS-1.
001812 REPORT-AGED-INVOICE-SC-1-102.
001820     IF AGE-OF-INVOICE GREATER THAN 30
001821         NEXT SENTENCE
001822     ELSE
001823         GO TO REPORT-AGED-INVOICE-SC-1-103.
001830     MOVE INVOICE-AMOUNT TO OVER-30-AMOUNT
001840     ADD INVOICE-AMOUNT TO CUSTOMER-OVER-30
001841     GO TO REPORT-AGED-INVOICE-SS-1.
001850 REPORT-AGED-INVOICE-SC-1-103.
001860     MOVE INVOICE-AMOUNT TO CURRENT-AMOUNT
001870     ADD INVOICE-AMOUNT TO CUSTOMER-CURRENT.
001880 REPORT-AGED-INVOICE-SS-1.
001890     WRITE AGED-TRIAL-LINE FROM AGED-LINE AFTER ADVANCING 1
001900         LINES.
001901 REPORT-AGED-INVOICE-EXIT.
```

## 7.2 Correlation of Generated COBOL to Structured Source

The CA-MetaCOBOL+ Translate-time options, RESEQ= and ID=, assist the programmer in tracing compiler errors back to the structured source code. Selection of these options ensures that source text which is not changed by macro processing has the same sequence number as it had originally. Also, this selection ensures that the replacement text produced by macros has the same sequence number as the source text which matched the macro prototype.

The ID= Translate-time option controls the contents of columns 73-80 of the output source records:

### Format:

$$\text{ID} = \left\{ \begin{array}{l} \text{name} \\ \text{*ID} \\ \text{*SEQ} \\ \text{*MC} \\ \text{*SAVE} \\ \text{*BLANK} \end{array} \right\}$$

See the CA-MetaCOBOL+ *User Guide* for a complete explanation of ID=.

The RESEQ= Translate-time option controls resequencing of the COBOL source output in columns 1-6. There are three operands available with RESEQ=:

### Format:

$$\text{RESEQ} = \left\{ \begin{array}{l} 0 \\ n \\ \text{NUM} \end{array} \right\}$$

Note that the concurrent usage of RESEQ=NUM and ID=*MC is the recommended approach.

When RESEQ=0, the expanded text has the same numbers as the source text, but the numbers may be out of sequence due to out-of-line code generation, and duplicate numbers may appear if the macros generate multiple lines of expanded text for one line of source text. Thus RESEQ=0 means no resequencing is selected. The user can trace compiler errors by using the compiler diagnostic's sequence number, finding the statement in error on the compiler listing, and using the original sequence number in columns 1-6 to locate the incorrect statement on the Input listing. To facilitate tracing compiler errors when RESEQ=0 is used, the source text should be numbered in ascending sequence with no duplicate numbers.

When RESEQ=*n*, CA-MetaCOBOL+ renumbers the entire generated program, and the user cannot trace errors back to the Input listing using numbers on the compiler listing. The letter *n* is a 1- to 3-digit integer greater than 0. The first COBOL source record output contains the number, *n*, right-aligned and left zero-filled, as the sequence number. Each succeeding COBOL output record contains sequence numbers incremented by *n*. To facilitate tracing compiler errors when RESEQ=*n* is used, use the ID=*MC Translate-time option.

When RESEQ=NUM, some, most, or all of the expanded text has different numbers than the source text because of the numbering increment used on the source text and the amount of code generated by the macros. When RESEQ=NUM or RESEQ=*n*, ID=*MC is useful because it places the Input listing line number in columns 73-80 of the expanded text. This permits the user to trace a compiler diagnostic by finding the statement in error on the compiler listing, and then using the number in columns 73-80 to locate the original input statement on the Input listing. To facilitate tracing compiler errors when RESEQ=NUM is used, the source text should be numbered in ascending sequence, with no duplicate numbers, a minimum numbering increment of 10, and a recommended numbering increment of 100.

To ensure the best numbering whatever options are used, each macro verb should be coded on a line separate from other macro verbs and COBOL verbs.

The example below, which contains an intended error, illustrates the use of the RESEQ=NUM and ID=*MC statements.

**Structured Source from the Input Listing**

```
   01535    001690    MOVE INVOICE-CUSTOMER TO AGED-LINE-CUSTOMER
   01536    001700    MOVE INVOICE-NO TO AGED-LINE-INVOICE
   01537    001710    MOVE INVOICE-DATE TO AGED-LINE-DATE
   01538    001720    SELECT FIRST ACTION
   01539    001730    WHEN AGE-OF-INVOICE IS GREATER THAN 90
-->01540    001740       MOVE INVOICE-AMOONT TO OVER-90-AMOUNT
   01541    001750       ADD INVOICE-AMOUNT TO CUSTOMER-OVER-90
   01542    001760    WHEN AGE-OF-INVOICE IS GREATER THAN 60
   01543    001770       MOVE INVOICE-AMOUNT TO OVER-60-AMOUNT
   01544    001780       ADD INVOICE-AMOUNT TO CUSTOMER-OVER-60
   01545    001790    WHEN AGE-OF-INVOICE IS GREATER THAN 30
   01546    001800       MOVE INVOICE-AMOUNT TO OVER-30-AMOUNT
   01547    001810       ADD INVOICE-AMOUNT TO CUSTOMER-OVER-30
   01548    001820    WHEN NONE ARE SELECTED
   01549    001830       MOVE INVOICE-AMOUNT TO CURRENT-AMOUNT
   01550    001840       ADD INVOICE-AMOUNT TO CUSTOMER-CURRENT
   01551    001850    ENDSELECT
```

**Generated COBOL Source from the Compilation Listing**

```
   001650 AGE-INVOICE-EXIT.
*MC01531
   001651 REPORT-AGED-INVOICE.
*MC01531
   001660     ADD 1 TO NUMBER-OF-LINES
*MC01532
   001670     PERFORM POSSIBLE-PAGE-BREAK THRU POSSIBLE-PAGE-BREAK-EXIT
*MC01533
   001680     MOVE SPACE TO AGED-LINE
*MC01534
   001690     MOVE INVOICE-CUSTOMER TO AGED-LINE-CUSTOMER
*MC01535
   001700     MOVE INVOICE-NO TO AGED-LINE-INVOICE
*MC01536
   001710     MOVE INVOICE-DATE TO AGED-LINE-DATE
*MC01537
   001730     IF AGE-OF-INVOICE GREATER THAN 90
*MC01539
   001731       NEXT SENTENCE
*MC01539
   001732     ELSE
*MC01539
   001733       GO TO REPORT-AGED-INVOICE-SC-1-101.
*MC01539
-->001740     MOVE INVOICE-AMOONT TO OVER-90-AMOUNT
-->*MC01540
   001750     ADD INVOICE-AMOUNT TO CUSTOMER-OVER-90
*MC01541
   001760     GO TO REPORT-AGED-INVOICE-SS-1.
*MC01542
   001761 REPORT-AGED-INVOICE-SC-1-101.
*MC01542
   001762     IF AGE-OF-INVOICE GREATER THAN 60
*MC01542
   001763       NEXT SENTENCE
*MC01542
   001764     ELSE
*MC01542
   001765       GO TO REPORT-AGED-INVOICE-SC-1-102.
*MC01542
   001770     MOVE INVOICE-AMOUNT TO OVER-60-AMOUNT
*MC01543
   001780     ADD INVOICE-AMOUNT TO CUSTOMER-OVER-60
*MC01544
   001790     GO TO REPORT-AGED-INVOICE-SS-1.
*MC01545
```

**Diagnostic:**

| **CARD** | **ERROR MESSAGE** | |
|---|---|---|
| 1740 | IKF3001I-E | INVOICE-AMOONT NOT DEFINED. DISCARDED. |

To correct this error, trace line 1740 to the generated COBOL source. The information found in columns 73-80 of line 1740 directs you to line 1540 of the structured source.

## 7.3 Verification at the Structured Source Level

With structured programs, it is possible to verify a program as it is being written. The high level modules in the program, controlling the path of program execution, are written and tested first. As successively lower level modules are completed, they are tested with, and added to, the already completed code. Then if errors are found in the program, they probably relate to the most recently added module. However, even this top-down development approach cannot guarantee that errors will not occur during development or later during maintenance.

You can do the following to verify structured programs:

- The Module Where Invoked Report produced by SPD tells where every module is defined and referenced. This report can be used to trace module use throughout the program.
- Translating with the RESEQ=NUM and ID=*MC Translate-time options should assist in providing a correlation between the line numbers shown in compiler diagnostics, Procedure Division maps, and the line numbers in the structured source.
- If checking a memory dump is required, no special procedures need to be followed. The programmer checks the dump for the location of the instruction that caused the abnormal termination. Once the location is determined, the PMAP or CLIST is used to get the sequence number of the source statement in question. The line number shown in the PMAP or CLIST list is then traced to the structured source (see Section 7.2).

If you think it is necessary to check the generated COBOL code when trying to determine the location of a problem, the modular structure of the program is maintained, as already noted, by the use of local paragraph-names and GO TO statements.

# 8. General Programming Verbs (GPV)

General Programming Verbs (GPV) are used to update, match, or merge files, initialize records and tables, access CA LIBRARIAN, manipulate bits, check PARM fields, and sort tables.

## 8.1 The ARRANGE Verb

The ARRANGE verb sorts a table into ascending or descending sequence:

**Format:**

```
ARRANGE      data-name-1

        [OCCURS   { data-name-2 } TIMES ]
                  { literal-1   }

        { ASCENDING  }  KEY IS   data-name-3
        { DESCENDING }

{ ENDARRANGE  }
{ END-ARRANGE }
```

**data-name-1**
represents the name of the one-dimensional table to be ARRANGEd, which may be qualified but not subscripted. The definition for *data-name-1* must include an OCCURS clause. KEY and INDEXED clauses are not essential to ARRANGE.

**OCCURS...TIMES**
Specifies the number of table entries to be sorted. If omitted, code is generated to sort all of the entries in the table. There are two uses for this clause:

- By itself, specification of this clause limits the sort arbitrarily to the first *n* entries.
- When the table definition includes a DEPENDING ON phrase, the specification of this clause should limit ARRANGE to the actual number of entries available. In this situation, *data-name-2* would be the same data-name coded in the DEPENDING ON phrase.

**data-name-2**
Represents the number of entries in the table to be ARRANGEd. It must be defined as an integer numeric elementary item that represents a positive integer, and it cannot refer to an index-name or index data item. If the table definition includes a DEPENDING ON phrase, *data-name-2* should be the same data-name coded in the DEPENDING ON phrase.

**literal-1**
Represents the number of entries in the table to be ARRANGEd. It must be a character string that represents a positive integer value, and it cannot refer to an index-name or index data item.

**ASCENDING**
Specifies that the table entries are to be ARRANGEd in ascending order; this is the default.

**DESCENDING**
Specifies that the table entries are to be ARRANGEd in descending order.

**KEY IS data-name-3**
Specifies the name of the sequence key field. The *data-name-3* may be qualified but not subscripted, and must be subordinate to *data-name-1*.

**Example:** In the following example, all entries in the SUMMARY-ENTRY table are ARRANGEd in descending sequence by SUMMARY-TOTAL. The second ARRANGE verb specifies that the entries in the DATA-ENTRY table are ARRANGEd in ascending sequence by DATA-KEY. The number of entries available for ARRANGEing is limited by the NUMBER-OF-ENTRIES value.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SUMMARY-TABLE.
    02  SUMMARY-ENTRY OCCURS . . .
        03  SUMMARY-TOTAL . . .
01  DATA-RECORD.
    02  NUMBER-OF-ENTRIES . . .
 . . .
    02  DATA-ENTRY OCCURS 10 TO 100
        TIMES DEPENDING ON
        NUMBER-OF-ENTRIES . . .
        03  DATA-KEY . . .
 . . .
PROCEDURE DIVISION.
    ARRANGE SUMMARY-ENTRY
        DESCENDING KEY IS SUMMARY-TOTAL
    ENDARRANGE
    ARRANGE DATA-ENTRY
        OCCURS NUMBER-OF-ENTRIES TIMES
        ASCENDING KEY IS DATA-KEY
    ENDARRANGE
```

**Notes:** ARRANGE is a required keyword.

ENDARRANGE/END-ARRANGE is a required keyword specifying the end of the ARRANGE verb.

## 8.2 The COLLATE Verb

The COLLATE verb may be used to control the updating of a master file from a transaction file, to match files for reporting purposes, or to merge two sequential files. COLLATE compares the keys of two sequentially accessed input files; subordinate processing is invoked based on the results of the comparison. COLLATE generates the COBOL logic required to OPEN, READ, and CLOSE the files.

```
COLLATE WITH [NO] REPORT
     PRIMARY FILE IS file-name-1
       { KEY IS   } data-name-1 [ ASCENDING  ]
       { KEYS ARE }             [ DESCENDING ]

                   [ data-name-2 [ ASCENDING  ] ]
                   [             [ DESCENDING ] ]

     SECONDARY FILE IS file-name-2
       { KEY IS   } data-name-3  [data-name-4] . . .   ASCENDING
       { KEYS ARE }

  { WHEN PRIMARY MATCHED                                       }
  {     PERFORM procedure-name [THRU procedure-name]           }
  {                                                            }
  { WHEN PRIMARY { UNMATCHED  }                                }
  {              { UN-MATCHED }                                }
  {                                                            }
  {     PERFORM procedure-name [THRU procedure-name]           }
  { WHEN SECONDARY { UNMATCHED  }                              }
  {                { UN-MATCHED }                              }
  {                                                            }
  {     PERFORM procedure-name [THRU procedure-name]           }
  [                                                            ]
  [ WHEN PRIMARY SEQUENCE ERROR                                ]
  [     PERFORM procedure-name [THRU procedure-name]           ]

  [ WHEN PRIMARY DUPLICATE ERROR                               ]
  [     PERFORM procedure-name [THRU procedure-name]           ]

  [ WHEN SECONDARY SEQUENCE ERROR                              ]
  [     PERFORM procedure-name [THRU procedure-name]           ]

  [ WHEN SECONDARY DUPLICATE ERROR                             ]
  [     PERFORM procedure-name [THRU procedure-name]           ]

{ ENDCOLLATE  }
{ END-COLLATE }
```

**PRIMARY FILE IS file-name-1**

Specifies the name of the primary input file. *File-name-1* must be defined as a sequential file or as a file that is accessed sequentially.

**KEY IS/KEYS ARE**

Defines the key or keys to be compared;  a maximum of ten may be defined.

**data-name-1**
Represents the name of the key within *file-name-1* that is compared with *data-name-3*.

**data-name-2**
Represents the name of the key within *file-name-1* that is compared with *data-name-4*.

**ASCENDING**
Specifies that the key from each file is compared in ascending sequence; this is the default.

**DESCENDING**
Specifies that the key from each file is compared in descending sequence.

**SECONDARY FILE IS file-name-2**
Specifies the name of the secondary input file: *file-name-2* must be defined as a sequential file or as a file that is accessed sequentially.

**data-name-3**
Represents the name of the key within *file-name-2* that is compared with *data-name-1*.

**data-name-4**
Represents the name of the key within *file-name-2* that is compared with *data-name-2*.

**WHEN PRIMARY MATCHED**
Is a condition that specifies procedures to be PERFORMed when the primary and secondary keys match.

**WHEN PRIMARY UNMATCHED/UN-MATCHED**
Is a condition that specifies procedures to be PERFORMed when the secondary key does not match the primary key.

**PERFORM procedure-name**
Specifies the name of the procedure to be PERFORMed when the results of the WHEN condition tests true.

**THRU procedure-name**
Specifies the name of the last procedure to be PERFORMed when the results of the WHEN condition tests true.

**WHEN SECONDARY UNMATCHED/UN-MATCHED**
Is a condition that specifies procedures to be PERFORMed when the primary key does not match the secondary key.

**WHEN PRIMARY SEQUENCE ERROR**
Is a condition that specifies procedures to be PERFORMed when a key sequence error is found within the primary file.

**WHEN PRIMARY DUPLICATE RECORD**
Is a condition that specifies procedures to be PERFORMed when a duplicate record is found within the primary file.

**WHEN SECONDARY SEQUENCE ERROR**
Is a condition that specifies procedures to be PERFORMed when a key sequence error is found within the secondary file.

**WHEN SECONDARY DUPLICATE RECORD**
Is a condition that specifies procedures to be PERFORMed when a duplicate record is found within the secondary file.

**Example:** In the following example, two files, SALES-REP-MASTER and IN-ORDER-FILE, are read. SALES-REP from the primary file is compared to IN-SALES-REP from the secondary file. If matching keys are found, the procedure SALES-REP-FOUND is PERFORMed, another IN-ORDER-FILE record is read, and the comparison repeated. If a match is not found, the procedure SALES-REP-NOT-FOUND is PERFORMed, another IN-ORDER-FILE record is read, and the comparison repeated. When both files are exhausted, control is passed to the next statement.

```
PERFORM BEGIN-ROUTINE
COLLATE WITH REPORT
    PRIMARY FILE IS SALES-REP-MASTER
        KEY IS SALES-REP
    SECONDARY FILE IS IN-ORDER-FILE
        KEY IS IN-SALES-REP
    WHEN PRIMARY MATCHED
        PERFORM SALES-REP-FOUND
    WHEN PRIMARY UNMATCHED
        PERFORM SALES-REP-NOT-FOUND
ENDCOLLATE
PERFORM SUMMARY
STOP RUN
```

**Notes:** COLLATE WITH REPORT are required keywords.

ENDCOLLATE/END-COLLATE is a required keyword specifying the end of the COLLATE verb.

At least one of the MATCHED/UNMATCHED WHEN conditions must be specified.

COLLATE controls the reading of the two input files and supports sequence checking.

## 8.3 The JCL-PARAMETERS Verb

The JCL-PARAMETERS verb generates code that checks the PARM field of the JCL EXEC statement for an operand that corresponds to a specified keyword. If detected, a data-item is modified. PARM list keywords and keywords connected to operands by the equal sign (=) are supported.

### Format:

```
JCL PARAMETERS
WHEN KEYWORD = keyword 1

MOVE { VALUE     }  TO data-name-1
     { literal-1 }

WHEN KEYWORD = keyword 2
[ MOVE { VALUE     }  TO data-name-2 ]  . . .
[      { Literal-1 }                 ]

[ WHEN ERROR                                                   ]
[     PERFORM procedure name-1 [THRU procedure name-2]         ]

{ ENDJCL-PARAMETERS  }
{ END-JCL-PARAMETERS }
```

**WHEN KEYWORD = keyword-n**
Specifies the keyword to be checked against the PARM field of the JCL EXEC statement. The *keyword-n* must be a character string containing up to 100 characters in length.

**MOVE...TO...**
Specifies that the JCL parameter VALUE or *literal-n* will be MOVEd to *data-name-n* when the WHEN KEYWORD condition tests true, i.e., a PARM operand matching *keyword-n* was detected.

**VALUE**
Specifies that the value on the PARM operand (KEYWORD=*value*) matching *keyword-n* is MOVEd to *data-name-1, -2*. If specified, *keyword-n* must assume that a corresponding PARM operand is followed by an equal sign and a value.

**literal-n**
Represents a literal or figurative constant to be MOVEd to *data-name-n.*

**data-name-n**
Represents the name of the data item to which the value of either VALUE or *literal-n* is MOVEd when the WHEN KEYWORD condition tests true.

**WHEN ERROR**
Is a condition that tests the validity of the PARM field.

**PERFORM procedure-name-1**
Specifies the name of the procedure to be PERFORMed when the WHEN ERROR condition tests true (no keywords were found or a required equal sign and value were omitted from a keyword).

**THRU procedure-name-2**
Specifies the name of the last procedure to be PERFORMed when the WHEN ERROR condition tests true (an error is found).

**Example:** In this example, assume the following JCL EXEC statement:

```
// EXEC PGM=pgm1,PARM=(SUMMARY,DATE=12/31/91)
```

The following input checks the PARM field for three keywords: DATE, LIST, and SUMMARY. If detected, the MOVE statement immediately following the true WHEN KEYWORD *condition* will be executed.

```
JCL-PARAMETERS
    WHEN KEYWORD = DATE
        MOVE VALUE TO END-OF-MONTH
    WHEN KEYWORD = LIST
        MOVE 'Y' TO LIST-SWITCH
    WHEN KEYWORD = SUMMARY
        MOVE 'Y' TO SUMMARY-SWITCH
ENDJCL-PARAMETERS

The resulting value for the data items specified on the MOVE
statements would then be:

END-OF-MONTH        12/31/91
LIST-SWITCH         N (default)
SUMMARY-SWITCH      Y
```

**Notes:** JCL-PARAMETERS is a required keyword.

ENDJCL-PARAMETERS/END-JCL-PARAMETERS is the required keyword specifying the end of the JCL-PARAMETERS verb.

All possible PARM operands must be defined within a single JCL-PARAMETERS verb; subsequent JCL-PARAMETERS verbs are ignored.

The JCL-PARAMETERS verb must be placed in the Procedure Division at the logical point of parameter interpretation.

Data items to be initialized must be user-defined.

## 8.4 FAIR Verbs

FAIR verbs support accessing CA LIBRARIAN, allowing the user to create a COBOL program which can retrieve any or all modules from a CA LIBRARIAN file.

Both tape and disk CA LIBRARIAN masters are supported. However, only one CA LIBRARIAN master file can be accessed in any execution of the COBOL program, and CA LIBRARIAN files cannot be updated by the generated program.

The FAIR COBOL copy book FAIRCBL must be available to the compiler. The user should be familiar with the FAIR interface and able to interpret the fields in the copy book which are initialized by FAIR.

All FAIR verbs are coded in the Procedure Division, and must be coded in the order presented; none may be omitted. An IF RETURN-CODE NOT = 0 must be coded to handle FAIR error conditions. An example showing the use of FAIR verbs is provided at the end of this section.

### 8.4.1 The OPEN-FAIR Verb

The OPEN-FAIR verb initializes CA LIBRARIAN for processing.

### Format:

```
OPEN-FAIR  [ DISK ]
           [ TAPE ]
```

**DISK/TAPE**
Specifies that FAIR is accessing a disk or tape master CA LIBRARIAN; DISK is the default.

**Note:** Tape master file support is not supported for CA LIBRARAN Release 3.9 and above.

**Notes:** OPEN-FAIR is a required keyword.

The OPEN-FAIR verb may be executed more than once.

For MVS systems, the OPEN processing can determine the type of file from the JCL in the following manner:

A MASTER DD card causes the disk CA LIBRARIAN file described to be readied for processing;

A CYCLE DD card causes the appropriate tape CA LIBRARIAN file to be readied;

A MASTIN DD card causes the tape CA LIBRARIAN file described to be readied;

Or an abnormal return occurs when JCL is omitted.

### 8.4.2 The START-FAIR Verb

The START-FAIR verb positions the CA LIBRARIAN file to the beginning of a module.

#### Format:

```
START-FAIR
    USING KEY { literal    }
              { identifier }
   [WITH [NO] -INC]
```

**USING KEY**
Defines the module to be read as specified by literal or identifier.

**literal**
Represents the name of the module to be read;  it must be non-numeric.

**identifier**
Represents the name of the module to be read;  it may be qualified and subscripted (indexed).

**WITH [NO] -INC**
Is specified for disk CA LIBRARIAN master files only.  If:

WITH -INC is specified, the -INC records are read, but the included text is not.

WITH NO -INC is specified, each -INC record is replaced by the included text.

Omitted, both the -INC and included text are read.

**Notes:** START-FAIR is a required keyword.

The START-FAIR verb may be executed more than once.  Each execution repositions the master file as specified.  It is possible for the file to be repositioned before the end of the current module is reached.

Once the method of handling -INC is established for a module, it cannot be changed.

## 8.4.3 The READ-FAIR Verb

The READ-FAIR verb places module records into a Working-Storage area.

### Format:

```
READ-FAIR
    [INTO identifier]
```

**INTO identifier**
Specifies the name of the area into which the record is placed. If omitted, the record is placed into FAIRCBL.

**Notes:** READ-FAIR is a required keyword.

Each execution of the READ-FAIR verb places the next record of a module into the named area or FAIRCBL.

## 8.4.4 The CLOSE-FAIR Verb

The CLOSE-FAIR verb closes the CA LIBRARIAN file.

### Format:

```
CLOSE-FAIR
```

**Notes:** The CLOSE-FAIR verb can be executed more than once, but is ignored if the file is already closed.

### 8.4.5 FAIR Verbs Example

The following example shows a program that might be used to list the preambles of all the CA-MetaCOBOL+ macro sets on a CA LIBRARIAN file.

```
*$COPY FAIR
 IDENTIFICATION DIVISION.
 PROGRAM-ID. PROLOGUE.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-370.
 OBJECT-COMPUTER. IBM-370.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT PRINT-FILE ASSIGN TO UT-S-SYSPRINT.
 DATA DIVISION.
 FILE SECTION.
 FD  PRINT-FILE        RECORDING MODE F
                       BLOCK CONTAINS 0 RECORDS
                       LABEL RECORD OMITTED
                       DATA RECORD PRINT-LINE.
 01  PRINT-LINE.
     02 FILLER         PIC X.
     02 PRINT-RECORD.
        03 FILLER      PIC X(6).
        03 PRINT-7     PIC X.
           88 COMMENT-LINE VALUES ARE '*' '/'.
        03 FILLER      PIC X(113).
 PROCEDURE DIVISION.
 MAIN.
     OPEN OUTPUT PRINT-FILE
     OPEN-FAIR
     IF RETURN-CODE NOT = 0
         PERFORM FAIR-ERROR
     ENDIF
     START-FAIR USING KEY SPACES
     IF RETURN-CODE NOT = 0
         PERFORM FAIR-ERROR
     ENDIF
     MOVE SPACES TO PRINT-LINE
     LOOP
         READ-FAIR INTO PRINT-RECORD
     UNTIL RETURN-CODE NOT = 0
         IF COMMENT-LINE
             WRITE PRINT-LINE
                 FROM RECRECD BEFORE ADVANCING 1 LINE
         ENDIF
     ENDLOOP
     PERFORM END-OF-JOB
     GOBACK
 END-OF-JOB.
     CLOSE-FAIR
     IF RETURN-CODE NOT = 0
         PERFORM FAIR-ERROR
     ENDIF
     CLOSE PRINT-FILE
 FAIR-ERROR.
     DISPLAY 'FAIR ERROR'
     CLOSE PRINT-FILE
     STOP RUN
```

## 8.5 Initialization Verbs

The two initialization verbs, CLEAR and RESET, initialize records and tables to move SPACEs or ZEROs to specified data-items.

### 8.5.1 The CLEAR Verb

The CLEAR verb initializes records and tables by generating code to move SPACE or ZERO to specified data-items that do not have a VALUE clause.

#### Format:

```
CLEAR identifier-1
     {THRU identifier-2 }
```

**identifier-1**

Represents the name of the first data item to which SPACE and ZERO are moved. CLEAR moves SPACE and ZERO beginning with *identifier-1* through the last elementary item of *identifier-1*, or *identifier-2* if the THRU clause is specified.

**THRU identifier-2**

Specifies the name of the last data item to which SPACE and ZERO are to be moved. If omitted, CLEAR moves SPACE and ZERO beginning with identifier-1 through the last elementary item of *identifier-1*.

**Example:** In the following example, CLEAR initializes the group item OUT-ORDER-RECORD which contains only one numeric field, OUT-AMOUNT:

```
DATA DIVISION.
FILE SECTION.
01  OUT-ORDER-RECORD.
    02  OUT-ORDER-DESCRP  PIC X(32).
    02  OUT-AMOUNT  PIC 999.
    02  FILLER . . .
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
CLEAR OUT-ORDER-RECORD
```

**The resulting output would be:**

```
MOVE SPACE TO OUT-ORDER-RECORD
MOVE ZERO TO OUT-AMOUNT
```

**Notes:** CLEAR is a required keyword.

Items having a REDEFINES clause or those subordinate to redefined items are not CLEARed, unless *identifier-1* is an item with a REDEFINES clause.

CLEAR generates MOVEs of SPACE and ZERO as follows:

- A MOVE SPACE is generated for each non-numeric or group item not subordinate to a group item already CLEARed by this execution and having no VALUE clause or subordinate items with a VALUE clause.
- A MOVE ZERO is generated for any numeric item with no VALUE clause.

## 8.5.2 The RESET Verb

The RESET verb initializes records and tables by generating code to move SPACE or ZERO to specified data-items.

### Format:

```
RESET identifier-1
    [THRU identifier-2]
```

**identifier-1**

Represents the name of the first data item to which SPACE and ZERO are moved. RESET moves SPACE and ZERO beginning with *identifier-1* through the last elementary item of *identifier-1*, or *identifier-2* if the THRU clause is specified.

**THRU identifier-2**

Specifies the name of the last data item to which SPACE and ZERO are to be moved. If omitted, RESET moves SPACE and ZERO beginning with *identifier-1* through the last elementary item of *identifier-1*.

**Example:** In the following example, RESET initializes the group item OUT-ORDER-RECORD which contains only one numeric field, OUT-AMOUNT:

```
DATA DIVISION.
FILE SECTION.
01  OUT-ORDER-RECORD.
    02  OUT-ORDER-DESCRP  PIC X(32).
    02  OUT-AMOUNT  PIC 999.
    02  FILLER . . .
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
RESET OUT-ORDER-RECORD
```

**The resulting output would be:**

```
MOVE SPACE TO OUT-ORDER-RECORD
MOVE ZERO TO OUT-AMOUNT
```

**Notes:** RESET is a required keyword.

RESET generates MOVEs of SPACE and ZERO as follows:

- Items having a REDEFINES clause or those subordinate to redefined items are not RESET, unless *identifier-1* is an item with a REDEFINES clause.
- A MOVE SPACE is generated for each non-numeric or group item not subordinate to a group item already CLEARed by this execution.
- A MOVE ZERO is generated for any numeric item.

## 8.6 Bit Manipulation Verbs

This section describes the bit manipulation verbs (COMPRESS, EXPAND, SET-BITS, and IF-BITS). An example showing the use of bit manipulation verbs is provided at the end of this section.

### 8.6.1 The COMPRESS Verb

The COMPRESS verb examines each byte of an 8-byte field and sets the corresponding bit in a 1-byte field. If the byte is the character '0', the bit is set to off; otherwise, the bit is set on (1).

#### Format:

```
COMPRESS identifier-1
    INTO identifier-2
```

**identifier-1**
Represents an 8-byte data item to be COMPRESSed.

**INTO identifier-2**
Specifies a 1-byte data item wherein the bits are set corresponding to the byte values of *identifier-1*.

**Notes:** COMPRESS is a required keyword.

The contents of *identifier-1* remain unchanged.

### 8.6.2 The EXPAND Verb

The EXPAND verb examines each bit of a 1-byte field and sets a corresponding byte in an 8-byte field to the character '0' if the bit is off or '1' if the bit is on.

#### Format:

```
EXPAND identifier-1
    INTO identifier-2
```

**identifier-1**
Represents a 1-byte data item to be EXPANDed.

**INTO identifier-2**
Specifies the name of an 8-byte data item wherein the bytes are set corresponding to the bit values of *identifier-1*.

**Notes:** EXPAND is a required keyword.

The contents of *identifier-1* remain unchanged.

### 8.6.3 The SET-BITS Verb

The SET-BITS verb sets the bits in a 1-byte field according to a mask.

#### Format:

```
SET-BITS identifier
   TO MASK mask
```

*identifier*
Represents a 1-byte field to be initialized.

**TO MASK** *mask*
Specifies a string of 8 characters corresponding to the bits in *identifier-1*. The valid characters are:

| | |
|---|---|
| **-** | the bit is not to be changed; |
| **0** | the bit is to be set to off; and |
| **1** | the bit is to be set to on. |

**Note:** SET-BITS is a required keyword.

### 8.6.4 The IF-BITS Verb

The IF-BITS verb examines the bits in a 1-byte field according to a mask, and a statement (or statements) is conditionally executed.

#### Format:

```
IF-BITS identifier
   MATCH MASK mask
   statement . . .
END-IF-BITS
```

**identifier**
Represents a 1-byte field.

**MATCH MASK mask**
Specifies a string of 8 characters corresponding to the bits in *identifier*. The valid characters are:

| | |
|---|---|
| **-** | the bit is not to be examined; |
| **0** | the bit must be off for the test to be true; and |
| **1** | the bit must be on for the test to be true. |

**statement**
Represents a COBOL statement to be performed when the result of the test is true.

**Note:** IF-BITS is a required keyword.

END-IF-BITS is the required keyword specifying the end of the IF-BITS verb.

## 8.6.5 Bit-Manipulation Verbs Example

The following example demonstrates a use for COMPRESS, EXPAND, SET-BITS, and IF-BITS for payroll deduction codes in both readable and encoded (or compressed) form.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  READABLE-RECORD.
    02  EXPANDED-DEDUCTION-CODES.
        03  ADDITIONAL-WITHHOLDING PIC X.
        03  UNION-DUES             PIC X.
        03  HEALTH-INSURANCE       PIC X.
        03  LIFE-INSURANCE         PIC X.
        03  BONDS                  PIC X.
        03  STOCK                  PIC X.
        03  CHARITY                PIC X.
        03  RETIREMENT             PIC X.
01  ENCODED-RECORD.
    02  COMPRESSED-DEDUCTION-CODES PIC X.
01  PAYROLL-DEDUCTIONS      COMP-3.
    02  UNION-DUES-AMOUNT          PIC S9(5)V99.
 . . .
PROCEDURE DIVISION.
 . . .
COMPRESS-DEDUCTION-CODES.
    COMPRESS EXPANDED-DEDUCTION-CODES
        INTO COMPRESSED-DEDUCTION-CODES
 . . .
EXPAND-DEDUCTION-CODES.
    EXPAND COMPRESSED-DEDUCTION-CODES
        INTO EXPANDED-DEDUCTION-CODES
 . . .
ADD-NEW-UNION-MEMBER.
*CHANGE DEDUCTIONS TO ADD UNION DUES
*AND MAKE INELIGIBLE FOR STOCK
    SET-BITS COMPRESSED-DEDUCTION-CODES
        TO MASK -1---0--
 . . .
UNION-DUES-DEDUCTIONS.
    MOVE ZERO TO UNION-DUES-AMOUNT
    IF-BITS COMPRESSED-DEDUCTION-CODES
        MATCH MASK -1000000
        MOVE +4.34 TO UNION-DUES-AMOUNT
    END-IF-BITS
```

# Appendix A - Structured Programming Facility Example

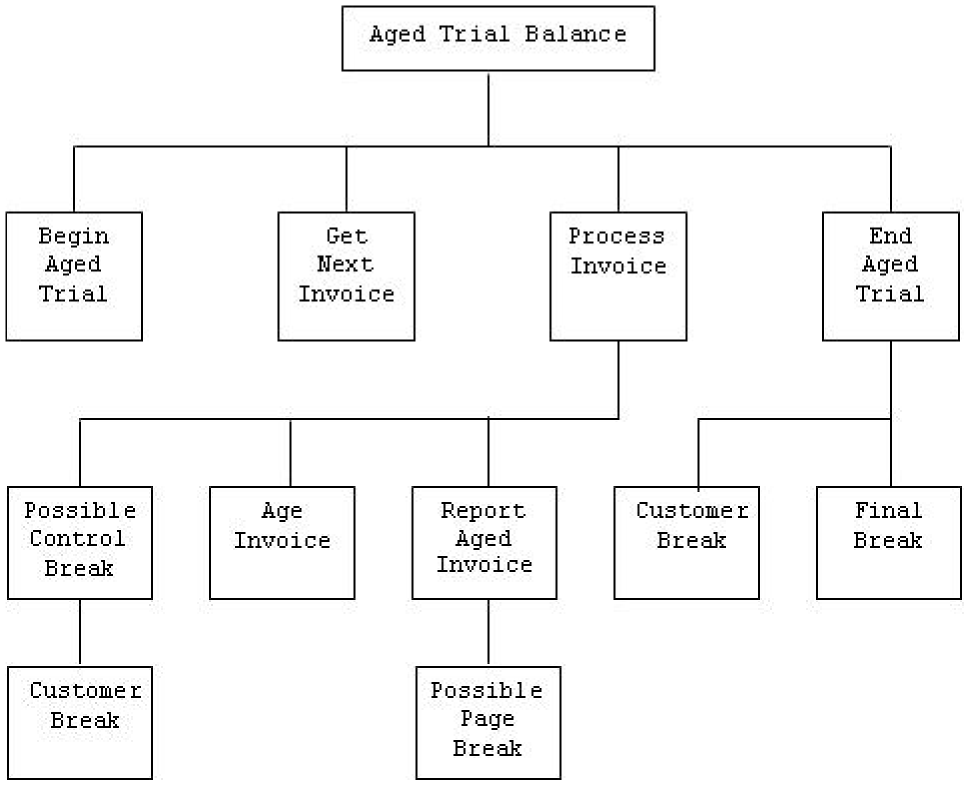
An Accounts Receivable Aged Trial Balance Reporting program is one of the classic programming problems presented in most programming courses. Because of this *classic* characteristic, it is used here to illustrate how to develop a problem statement into a fully operational program.

## A.1 Problem Statement

The program is to read an Accounts Receivable master file (in sequence by invoice number within customer id), and produce a report showing customers who have outstanding balances. These balances are to be shown as *current*, and as *over 30*, *over 60*, and *over 90* days delinquent.

As a first step in the design process, the following block diagram shows the functional processing steps required.

From the diagram, the program can be written directly in structured code, as shown in the following pages. Notice how each processing step is clearly defined and has only one entry point and one exit point--making the program easy to read, verify, and maintain.



## A.2 Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  AGEDTB.
AUTHOR.  BLUE TEAM.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  IBM-370.
OBJECT-COMPUTER.  IBM-370.
SPECIAL-NAMES.
    C01 IS TO-TOP-OF-PAGE.

INPUT-OUTPUT SECTION.
FILE-CONTROL.

    SELECT INVOICE-FILE
        ASSIGN TO UT-S-INVOICE.

    SELECT AGED-TRIAL-REPORT
        ASSIGN TO UT-S-SYSPRINT.

DATA DIVISION.
FILE SECTION.

FD  INVOICE-FILE
       RECORDING MODE IS F
       LABEL RECORDS ARE STANDARD
       BLOCK CONTAINS 10 RECORDS
       DATA RECORD IS INVOICE-RECORD.

01  INVOICE-RECORD.
    02  INVOICE-CUSTOMER      PIC X(7).
    02  INVOICE-NO            PIC X(7).
    02  INVOICE-DATE          PIC X(8).
    02  INVOICE-YEAR-DAYS.
        03  INVOICE-YEAR     PIC 99.
        03  INVOICE-DAYS     PIC 999.
    02  INVOICE-AMOUNT        PIC S9(9)V99  COMP-3.

FD  AGED-TRIAL-REPORT
        RECORDING MODE IS F
        LABEL RECORDS ARE OMITTED
        DATA RECORD IS AGED-TRIAL-LINE.

01  AGED-TRIAL-LINE           PIC X(121).
```

```
WORKING-STORAGE SECTION.
77  AGE-OF-INVOICE            PIC S9(5) COMP-3.
77  NUMBER-OF-LINES           PIC S9(3) COMP-3 VALUE +50.
77  MAXIMUM-LINES             PIC S9(3) COMP-3 VALUE +50.
77  NUMBER-OF-PAGES           PIC S9(5) COMP-3 VALUE ZERO.

01  CURRENT-YEAR-DAYS.
    O2  CURRENT-YEAR          PIC 99.
    02  CURRENT-DAYS          PIC 999.

01  CUSTOMER-TOTALS.
    02  CUSTOMER-CURRENT      PIC S9(9)V99 COMP-3 VALUE ZERO.
    02  CUSTOMER-OVER-30      PIC S9(9)V99 COMP-3 VALUE ZERO.
    02  CUSTOMER-OVER-60      PIC S9(9)V99 COMP-3 VALUE ZERO.
    02  CUSTOMER-OVER-90      PIC S9(9)V99 COMP-3 VALUE ZERO.

01  FINAL-TOTALS.
    02  FINAL-CURRENT         PIC S9(9)V99 COMP-3 VALUE ZERO.
    02  FINAL-OVER-30         PIC S9(9)V99 COMP-3 VALUE ZERO.
    02  FINAL-OVER-60         PIC S9(9)V99 COMP-3 VALUE ZERO.
    02  FINAL-OVER-90         PIC S9(9)V99 COMP-3 VALUE ZERO.

01  HEADING-LINE.
    02  FILLER                PIC X VALUE SPACE.
    02  HEADING-DATE          PIC X(8).
    02  FILLER                PIC X(26) VALUE SPACE.
    02  FILLER                PIC X(25)
        VALUE 'AGED TRIAL BALANCE REPORT'.
    02  FILLER                PIC X(26) VALUE SPACE.
    02  FILLER                PIC X(5) VALUE 'PAGE'.
    02  HEADING-PAGE          PIC Z(5).

01  SUB-HEADING-LINE-1.
    02  FILLER                PIC X(28)
        VALUE ' CUSTOMER INVOICE   INVOICE'.
    02  FILLER                PIC X(28)
        VALUE '*--------------------------'.
    02  FILLER                PIC X(12) VALUE 'AGED AMOUNTS'.
    02  FILLER                PIC X(28)
        VALUE '--------------------------*'.

01  SUB-HEADING-LINE-2
    02  FILLER                PIC X(37)
        VALUE '   NUMBER  NUMBER     DATE'.
    02  FILLER                PIC X(17) VALUE 'CURRENT'.
    02  FILLER                PIC X(17) VALUE 'OVER 30'.
    02  FILLER                PIC X(17) VALUE 'OVER 60'.
    02  FILLER                PIC X(17) VALUE 'OVER 90'.
```

```
CUSTOMER-BREAK.
    ADD 3 TO NUMBER-OF-LINES
    MOVE 'CUSTOMER' TO TOTAL-ID
    MOVE PREVIOUS-CUSTOMER TO TOTAL-CUSTOMER
    MOVE CUSTOMER-CURRENT TO TOTAL-CURRENT
    MOVE CUSTOMER-OVER-30 TO TOTAL-OVER-30
    MOVE CUSTOMER-OVER-60 TO TOTAL-OVER-60
    MOVE CUSTOMER-OVER-90 TO TOTAL-OVER-90
    WRITE AGED-TRIAL-LINE FROM TOTAL-LINE
        AFTER ADVANCING 2 LINES
    MOVE SPACE TO AGED-TRIAL-LINE
    WRITE AGED-TRIAL-LINE AFTER ADVANCING 1 LINES
    ADD CUSTOMER-CURRENT TO FINAL-CURRENT
    ADD CUSTOMER-OVER-30 TO FINAL-OVER-30
    ADD CUSTOMER-OVER-60 TO FINAL-OVER-60
    ADD CUSTOMER-OVER-90 TO FINAL-OVER-90
    MOVE ZERO TO CUSTOMER-CURRENT CUSTOMER-OVER-30
        CUSTOMER-OVER-60 CUSTOMER-OVER-90

FINAL-BREAK.
    MOVE 'FINAL' TO TOTAL-ID
    MOVE SPACE TO TOTAL-CUSTOMER
    MOVE FINAL-CURRENT TO TOTAL-CURRENT
    MOVE FINAL-OVER-30 TO TOTAL-OVER-30
    MOVE FINAL-OVER-60 TO TOTAL-OVER-60
    MOVE FINAL-OVER-90 TO TOTAL-OVER-90
    WRITE AGED-TRIAL-LINE FROM TOTAL-LINE
        AFTER ADVANCING 1 LINES

POSSIBLE-PAGE-BREAK.
    IF NUMBER-OF-LINES IS GREATER THAN MAXIMUM-LINES
        MOVE 1 TO NUMBER-OF-LINES
        ADD 1 TO NUMBER-OF-PAGES
        MOVE NUMBER-OF-PAGES TO HEADING-PAGE
        WRITE AGED-TRIAL-LINE FROM HEADING-LINE
            AFTER ADVANCING TO-TOP-OF-PAGE
        WRITE AGED-TRIAL-LINE FROM SUB-HEADING-LINE-1
            AFTER ADVANCING 2 LINES
        WRITE AGED-TRIAL-LINE FROM SUB-HEADING-LINE-2
            AFTER ADVANCING 1 LINES
        MOVE SPACES TO AGED-TRIAL-LINE
        WRITE AGED-TRIAL-LINE AFTER ADVANCING 1 LINES
    ENDIF
```

# Appendix B - SPP-Generated Data-Names and Procedure-Names

CA-MetaCOBOL+ translation of a structured source program under control of the Structured Programming Processor (SPP) produces a standard COBOL source program. The generated COBOL source contains additional data descriptions in the Working-Storage Section of the Data Division, and includes local procedure-names and references to GO TO and PERFORM statements.

The following sections describe the various categories of generated data-names and procedure-names to aid COBOL program understanding and source level verification. Section B.1 describes each of the SPP-generated data-names. Section B.2 describes each of the SPP-generated procedure-names that are module-oriented. The construct-oriented procedure-names are described in Section B.3.

## B.1 SPP-Generated Data-Names

The following describes each of the SPP-generated data-names:

*condition-name*
: is a level-88 item generated by a control variable definition (FLAG). It must be unique in the first 25 characters.

**ZSPP-***condition-name*
: The control variable associated with *condition-name*, and generated by a control variable definition (FLAG). If *condition-name* exceeds 25 characters in length, excess characters are truncated from the right.

**ZSPPSEL-***mmm-ccc*

is a level-77 control counter required by SELECT EVERY and SELECT LEADING postscript analysis.

*mmm* a 1- to 3-digit representation of the serial number of the module, incremented by the Procedure Division header, the DECLARATIVES header, ENTRY statements, section headers, and module headers.

*ccc* a 1- to 3-digit representation of the serial number of the construct (IF, SELECT, SEARCH, and LOOP) within the module.

## B.2 SPP-Generated, Module-Oriented Procedure-Names

The following describes each of the SPP-generated procedure-names that are module-oriented:

*module-name***-[SECTION]**

The *module-name* or *section-name* defined in the structured source. It must be unique in the first 19 characters.

*module-name***-EXIT**

Paragraph-name generated at the end of the module. Invoking DO *module-name* and PERFORM *module-name* statements become PERFORM *module-name* THRU *module-name*-EXIT in the generated COBOL. The *module-name* is the current paragraph-name.

Paragraph-name generated following the last module of a section; *module-name* is the current section-name.

If *module-name* exceeds 25 characters in length, excess characters are truncated from the right.

*module-name***-PGRM**

Paragraph-name generated ahead of an EXIT PROGRAM statement. If *module-name* exceeds 25 characters in length, excess characters are truncated from the right.

*module-name***-SECG**

Paragraph-name generated following the first module of a section and containing a GO TO to the section exit. If *module-name* exceeds 25 characters in length, excess characters are truncated from the right.

## B.3 SPP-Generated, Construct-Oriented Procedure-Names

Local procedure-names and references to GO TO and PERFORM statements are generated by each IF, LOOP, SEARCH, and SELECT construct within each module.

Each local procedure-name generated takes the form:

*module-name-XX-nnn*[-*sss*]

**module-name**

The *Module-name* must be unique in the first 19 characters. If the module-name exceeds 19 characters in length, excess characters are truncated from the right.

| | |
|---|---|
| *XX* | a two-character, alphabetic code that defines the construct type and component of the construct. |
| *nnn* | a 1-3 digit representation of the sequence number of the construct within the module. |
| *sss* | a unique, three-digit serial number of the local procedure-name within the construct. |

The following describes each of the SPP-generated procedure-names which are construct-oriented:

*module-name*-**FF-***nnn*

Procedure-name representing the beginning of the process subordinate to a LOOP...TIMES or LOOP...VARYING construct. Refer to *module-name*-FX-*nnn* below.

*module-name*-**FX-***nnn*

Procedure-name representing the end of the process subordinate to a LOOP...TIMES or LOOP...VARYING construct. The construct is replaced by a generated PERFORM *module-name*-FF-*nnn* THRU *module-name*-FX-*nnn* statement.

*module-name*-**HE-***nnn-sss*

Procedure-name representing the beginning of a process subordinate to a WHEN clause of a SEARCH construct. Refer to *module-name*-HF-*nnn-sss* below.

*module-name*-**HF-***nnn-sss*

Procedure-name representing the end of a process subordinate to a WHEN clause of a SEARCH construct. The process is replaced by a generated PERFORM *module-name*-HE-*nnn-sss* THRU *module-name*-HF-*nnn-sss* statement.

*module-name*-**HX-***nnn*

Procedure-name generated at the end of a SEARCH construct, and representing a bypass around the PERFORMed processes defined above.

*module-name*-**IF-***nnn-sss*

Procedure-name representing the beginning of the false path (ELSE) of an IF construct.

*module-name*-**IX-***nnn*

Procedure-name representing the end of an IF construct. A GO TO *module-name*-IX-*nnn* is generated at the end of the true path.

*module-name*-**LP-***nnn*

Procedure-name representing the head of a LOOP...WHILE, LOOP...LEAVE WHEN, or LOOP...UNTIL construct. A GO TO *module-name*-LP-*nnn* is generated at the location of the ENDLOOP.

*module-name*-**LX-***nnn*

Procedure-name representing the exit of a LOOP construct, placed following the location of the ENDLOOP. A GO TO *module-name*-LX-*nnn* is generated subordinate to the WHILE, LEAVE WHEN, or UNTIL condition, and at the location of an ESCAPE command.

*module-name*-**SC-***nnn-sss*

Procedure-name representing the next WHEN condition and first postscript in a SELECT...FOR or SELECT FIRST construct, the next WHEN condition or postscript in a SELECT EVERY construct, and the next postscript in a SELECT LEADING construct. The false path of the preceding WHEN condition contains a GO TO the subsequent *module-name*-SC-*nnn-sss*.

*module-name*-**SS-***nnn*

Procedure-name representing the WHEN ANY postscript in a SELECT...FOR or SELECT FIRST construct, or the first postscript in a SELECT LEADING construct. In a SELECT...FOR or SELECT FIRST, a GO TO *module-name*-SS-*nnn* is generated following the process in the true path. In a SELECT LEADING, a GO TO *module-name*-SS-*nnn* is generated for the false path of all WHEN conditions.

*module-name*-**SX-***nnn*

Procedure-name representing the end of a SELECT...FOR or SELECT FIRST construct. A GO TO *module-name*-SX-*nnn* is generated at the end of the first postscript.

# Appendix C - Diagnostics

Diagnostics described in this appendix are NOTE messages that are produced by the SPD, HLF, SPP, SPS, GPV, and SP2 macro sets. NOTE diagnostics contained with the above macro sets begin with a prefix, as follows:

*xxxnnc*

*xxx* represents the diagnostic abbreviation:

- **GPV** for General Programming Verbs.
- **HLF** for High-Level Formatter (HLF).
- **SPD** for Structured Programming Documentor (SPD).
- **SPP** for Structured Programming Processor (SPP).
- **SPS** for Structured Programming Stub (SPS) Generator.
- **SPV** for Structured Programming using VS COBOL II (SP2) compilers.

*nn* represents the diagnostic number, which is numeric and unique within *xxx*.

*c* represents a severity indicator:

- **E** an error requiring correction and retranslation, associated with a condition code of 12.
- **W** a warning, possibly requiring correction and retranslation, associated with a condition code of 8.
- **A** an advisory comment, associated with a condition code of 4.

NOTE diagnostics containing prefixes other than those given above are associated with other macro sets described in their respective CA-MetaCOBOL+ technical documentation.

To determine the cause of and corrective action for FATAL ERROR, ERROR, and WARNING diagnostics issued by CA-MetaCOBOL+, refer to the CA-MetaCOBOL+ *User Guide*.

## C.1 GPV Diagnostics

**GPV01E** **DATA-NAME "*data-name*" IS UNDEFINED**

**Explanation:** Either "*Data-name*" is misspelled, or the definition has been omitted from the program.

**Action:** Correct the spelling if incorrect or add the data definition to the program if omitted. Retranslate the program.

**GPV02E** **DATA-NAME "*data-name*" IS NOT ALPHA-NUMERIC**

**Explanation:** "*Data-name*" is not an alphanumeric item.

**Action:** "*Data-name*" must be an alphanumeric item; correct and retranslate the program.

**GPV03E** **DATA-NAME "*data-name*" LENGTH INCORRECT**

**Explanation:** "*Data-name*" is either too short or too long.

**Action:** Only an 8-byte data item is acceptable; correct and retranslate the program.

**GPV04E** **MASK "*mask*" LENGTH INCORRECT**

**Explanation:** "*Mask*" is either too short or too long.

**Action:** A mask must be 8 characters in length; correct the mask and retranslate the program.

**GPV05E** **MASK "*mask*" INVALID, CONTAINS NO ZERO OR ONE**

**Explanation:** "*Mask*" is invalid.

**Action:** A mask must contain at least one zero (0) or one integer one (1). Correct the mask and retranslate the program.

**GPV06E** **LOAD GPV MACROS AFTER SPP MACROS**

**Explanation:** The GPV macros have been loaded prior to the SPP macros.

**Action:** In order for the GPV macros to function correctly with the SPP macros, the SPP macros must be loaded before the GPV macros. Rearrange the translator-directing statement that loads the macros and retranslate the program.

**GPV07E** "*word*" **IS UNDEFINED**

**Explanation:** "*Word*" is undefined.

**Action:** Examine the context in which "*word*" is used and correct it to either a data-name or literal. Then retranslate the program.

**GPV10E** **DATA-NAME** "*data-name-1*" **FOLLOWS** "*data-name-2*"

**Explanation:** "*Data-name-1*" and "*data-name-2*" are not specified in "top down" order.

**Action:** The data-names in a CLEAR or RESET verb must proceed "top down." Reverse "*data-name-1*" and "*data-name-2*" and retranslate the program.

**GPV12E** "*CLEAR/RESET*" **SYNTAX INVALID, CHECK QUALIFICATION OR THRU**

**Explanation:** The verb syntax is incorrect.

**Action:** Either the data-name qualification is in error (OF or IN misspelled or omitted), or THRU is misspelled or omitted. Determine the cause of the error, correct it, and retranslate the program.

**GPV13E** **SUBSCRIPTED DATA-NAMES ARE NOT VALID**

**Explanation:** Subscripting of the data-names in CLEAR and RESET verbs is invalid.

**Action:** Remove the subscript(s) and retranslate the program.

**GPV14W** **CLEAR INEFFECTIVE, NO MOVES GENERATED**

**Explanation:** The CLEAR verb specified will not alter any data-names; they all have initialization values.

**Action:** Check the intent of the verb, and correct and retranslate the program.

**GPV15E** **DATA-NAME** "*data-name*" **MUST BE DATA ITEM**

**Explanation:** "*Data-name*" is not a valid data-name.

**Action:** Correct the data-name in error and retranslate the program.

**GPV16E** **MORE THAN 200 "REDEFINES," INCREASE OCCURS FOR &V@REDEFINES**

**Explanation:** The program contains more than 200 REDEFINES clauses, exceeding the size of a macro table.

**Action:** Locate the macro variable &V@REDEFINES in the macro code and increase the number of occurrences to an appropriately larger value Retranslate the program.

**GPV17W** "*word*" **DISCARDED**

**Explanation:** "*Word*" is superfluous or misplaced; it has been discarded.

**Action:** Check the verb for other errors, and correct and retranslate the program.

**GPV18E** **COLLATE ENDS WITH VERB,** "*verb*"

**Explanation:** END-COLLATE has been omitted or misspelled.

**Action:** Correct and retranslate the program.

**GPV19E** **COLLATE ENDS WITH MACRO,** "*macro-name*"

**Explanation:** END-COLLATE has been omitted or misspelled.

**Action:** Correct and retranslate the program.

**GPV20E** **COLLATE PRIMARY FILE UNDEFINED**

**Explanation:** The COLLATE primary file-name has been omitted or misspelled.

**Action:** Add the file-name definition to the program or correct its spelling; retranslate the program.

**GPV21E** **COLLATE PRIMARY KEY(S) UNDEFINED**

**Explanation:** The COLLATE primary key(s) has not been defined.

**Action:** At least one primary key must be defined. Correct the verb and retranslate the program.

**GPV22E** **COLLATE SECONDARY FILE UNDEFINED**

**Explanation:** The COLLATE secondary file-name has been omitted or misspelled.

**Action:** Add the file-name definition to the program or correct its spelling; retranslate the program.

**GPV23E** **COLLATE SECONDARY KEY(S) UNDEFINED**

**Explanation:** The COLLATE secondary key(s) has not been defined.

**Action:** At least one secondary key must be defined. Correct the verb and retranslate the program.

**GPV24E** **COLLATE KEY TALLYS DO NOT AGREE**

**Explanation:** The number of keys specified for the primary and secondary files does not agree.

**Action:** Check the verb for other errors, and correct and retranslate the program.

**GPV25E** **COLLATE KEYS, "***data-name-1***" AND "***data-name-2***", ARE NOT COMPATIBLE**

**Explanation:** "*Data-name-1*" AND "*data-name-2*" are not compatible keys.

**Action:** The "class" for matching keys must be the same. Correct the verb and retranslate the program.

**GPV26E** **REQUIRED COLLATE CONDITION OMITTED**

**Explanation:** A required COLLATE condition has been omitted.

**Action:** At least one of the three required conditions of the COLLATE verb must be specified. Correct the verb and retranslate the program.

**GPV27E** **ONLY 1 JCL-PARAMETERS ALLOWED**

**Explanation:** Two or more JCL-PARAMETERS verbs have been specified.

**Action:** Only one JCL-PARAMETERS verb may be specified in a program. Review the code and combine the JCL-PARAMETERS verbs; retranslate the program.

**GPV29E** **JCL-PARAMETERS ENDS WITH VERB,** "*verb*"

**Explanation:** END-JCL-PARAMETERS has been misspelled or omitted.

**Action:** Correct the statement and retranslate the program.

**GPV30E** **JCL-PARAMETERS ENDS WITH MACRO,** "*macro-name*"

**Explanation:** END-JCL-PARAMETERS has been misspelled or omitted.

**Action:** Correct the statement and retranslate the program.

**GPV31E** **JCL-PARAMETERS KEYWORDS OMITTED**

**Explanation:** One or more JCL-PARAMETERS keywords have been omitted.

**Action:** At least one keyword must be specified. Correct the statement and retranslate the program.

**GPV32E** **JCL-PARAMETERS KEYWORD INITIALIZATION,** "*data-name*"**, IS UNDEFINED**

**Explanation:** "*Data-name*" is either misspelled or omitted from the program.

**Action:** Correct the statement and retranslate the program.

**GPV33E** **DATA-NAME FOR JCL-PARAMETERS KEYWORD,** "*data-name*"**, IS UNDEFINED**

**Explanation:** "*Data-name*" is either misspelled or omitted from the program.

**Action:** Correct the statement and retranslate the program.

**GPV34E** **DATA-NAME AND INITIALIZATION FOR JCL-PARAMETERS KEYWORD** "*keyword*"**, ARE NOT COMPATIBLE**

**Explanation:** The data-name and the initialization value are inconsistent.

**Action:** Correct the data definition or change the initialization value; retranslate the program.

**GPV35W** **ARRANGE ENDS WITH VERB** "*verb*"

**Explanation:** END-ARRANGE has been misspelled or omitted.

**Action:** Correct the statement and retranslate the program.

**GPV36E** **ARRANGE ENDS WITH MACRO,** "*macro-name*"

**Explanation:** END-ARRANGE has been misspelled or omitted.

**Action:** Correct the statement and retranslate the program.

**GPV37E** **ARRANGE TABLE NAME,** "*data-name*"**, IS UNDEFINED**

**Explanation:** "*Data-name*" is either misspelled or omitted from the program.

**Action:** Correct the statement and retranslate the program.

**GPV38E** **ARRANGE TABLE NAME,** "*data-name*"**, CANNOT BE SUBSCRIPTED**

**Explanation:** "*Data-name*" is subscripted.

**Action:** "*Data-name*" may be qualified, but not subscripted, and its definition must include an OCCURS clause. Correct the statement and retranslate the program.

**GPV39E** **ARRANGE TABLE NAME, "*data-name*", HAS NO OCCURS**

**Explanation:** The definition for "*data-name*" does not include an OCCURS clause.

**Action:** Correct the definition and retranslate the program.

**GPV40E** **ARRANGE OCCURS, "*data-name*", IS UNDEFINED**

**Explanation:** "*Data-name*" is either misspelled or omitted from the program.

**Action:** Correct the statement and retranslate the program.

**GPV41E** **ARRANGE OCCURS, "*value*", MUST BE INTEGER**

**Explanation:** The OCCURS value (the number of table entries to be sorted) is not an integer value.

**Action:** Correct the statement and retranslate the program.

**GPV42E** **ARRANGE KEY NAME, "*data-name*", IS UNDEFINED**

**Explanation:** "*Data-name*" is either misspelled or omitted from the program.

**Action:** Correct the statement and retranslate the program.

**GPV43E** **ARRANGE KEY NAME, "*data-name*", CANNOT BE SUBSCRIPTED**

**Explanation:** The key name "*data-name*" is subscripted.

**Action:** "*Data-name*" may be qualified, but not subscripted, and must be part of the table definition. Correct the statement and retranslate the program.

**GPV44E** **ARRANGE KEY NAME, "*data-name-1*", IS NOT SUBORDINATE TO "*data-name-2*"**

**Explanation:** The key "*data-name-1*" is not subordinate to "*data-name-2*".

**Action:** The key "*data-name-1*" must be subordinate to the one-dimensional table "*data-name-2*" to be ARRANGEd. Correct the statement and retranslate the program.

## C.2 HLF Diagnostics

**HLF001E** **VALUE OF UPSI1 TRANSLATE-TIME OPTION INVALID.**

**Explanation:** The UPSI1 option must be set to **C** or **S** to indicate whether the program to be formatted is a COBOL or SP program.

**Action:** Correct and rerun.

**HLF101W** "*statement*" **CONSTRUCT BEGINNING ON LINE nnnn DOES NOT END WITH A SCOPE TERMINATOR.**

**Explanation:** A scope terminator is required by the indicated statement.

**Action:** Enter a valid scope terminator and rerun.

**HLF102E** "*word*" **DOES NOT HAVE A CORRESPONDING CONSTRUCT. ACTIVE CONSTRUCTS ARE: List of constructs.**

**Explanation:** A word recognized by HLF as a postscript (e.g., UNTIL or WHEN), scope terminator (e.g., ENDIF or ENDSEARCH), or condition (e.g., ON OVERFLOW or IF SIZE ERROR) does not have a construct associated with it. This error can be caused by incorrect nesting.

**Action:** Refer to the list of active constructs. Place the specified word or clause within the proper construct and rerun.

**HLF103E** **DUPLICATE CONDITION CLAUSE FOUND FOR** "*construct*" **STATEMENT.**

**Explanation:** Two clauses are associated with the same construct. This error may be caused by incorrect nesting.

**Action:** Review any nesting errors, such as misplaced scope terminators. Correct the error and rerun.

**HLF201W** **ENCOUNTERED SP FACILITY CONDITION IN COBOL PROGRAM.**

**Explanation:** An SP Facility condition, such as IF SIZE ERROR and IF NOT OVERFLOW, has been encountered, but COBOL formatting has been specified.

**Action:** If a SP Facility formatting is intended, specify UPSI1=S. If a COBOL formatting is intended, change SP Facility conditions to COBOL conditions.

**HLF301E** **MACRO SET HLF MUST BE LOADED BEFORE SPD.**

**Explanation:** SPD must be loaded after HLF.

**Action:** Correct order of macro input and re-run.

**HLF302E** **LOCAL DATA FROM "START DATA" NOT TERMINATED WITH "END DATA".**

**Explanation:** Local data in an SP program may not contain a verb, high-level statement, or paragraph name before the END DATA scope terminator.

**Action:** Determine whether the scope terminator is missing or has been coded incorrectly. Correct and rerun.

**HLF303W** **ENCOUNTERED COBOL CONDITION "*condition*" IN SP FACILITY PROGRAM.**

**Explanation:** A COBOL condition, such as ON OVERFLOW or SIZE ERROR, has been encountered, but SP Facility formatting has been specified.

**Action:** If a COBOL formatting is intended, specify UPSI1=C. If SP Facility formatting is intended, change the COBOL conditions to SP Facility conditions.

**HLF304W** **"ON" STATEMENT ENCOUNTERED IN SP FACILITY PROGRAM.**

**Explanation:** The ON debugging statement is not allowed in SP programs.

**Action:** If a COBOL formatting is intended, change the UPSI1 option to UPSI1=C. If SP Facility formatting is desired, the ON statement must be removed from the program.

**HLF401E** **"END-EXEC" NOT ENCOUNTERED IN PREVIOUS "EXEC-CICS".**

**Explanation:** A statement has interrupted formatting of an EXEC-CICS because it was encountered prior to END-EXEC.

**Action:** Move any non-CICS statements out of the EXEC-CICS logic. Insert EXEC-CICS if it is missing.

**HLF402W** **"EXEC-CICS" FOLLOWED IMMEDIATELY BY "END-EXEC".**

**Explanation:** The EXEC-CICS statement is empty.

**Action:** Either remove the EXEC-CICS and END-EXEC syntax or insert any missing CICS statements.

## C.3 SPD Diagnostics

**SPD01E** **MODULE TABLE FULL**

**Explanation:** SPD supports processing of 500 unique module-names. If more than 500 unique module-names are found, processing terminates.

**Action:** The module table, as defined in SPDCOB, should be enlarged to allow more entries.

**SPD02E** **CHAIN TABLE FULL**

**Explanation:** SPD supports processing of 7000 invoking processes (DO, PERFORM, and CALL). If the maximum is exceeded, processing terminates.

**Action:** The chain table, as defined in SPDCOB, should be enlarged to allow more entries.

**SPD03E** **TOO MANY UNDEFINED MODULE NAMES**

**Explanation:** More than 50 undefined module-names or stubs have been encountered in the program. If both reports are requested, only the Module Where Invoked Report is produced.

**Action:** None. The Module Hierarchy Report is not produced when more than 50 modules are undefined.

**SPD04E** **INPUT RECORD ERROR**

**Explanation:** SPDCOB does not recognize the attributes of the input records passed to it. Processing terminates. No reports are produced.

**Action:** Review the JCL to ensure that the correct temporary data set is passed to the step executing.

**SPD05E** **NO HIERARCHY TO REPORT**

**Explanation:** No procedure names or section headers were found in the program.

**Action:** Review program intended for documentation.

**SPD06E** **NESTING LIMIT IS 99, HIERARCHY REPORT TERMINATED**

**Explanation:** The program has modules nested to a depth greater than 99. The Module Hierarchy Report is terminated.

**Action:** Review program structure.

**SPD07E** **SORTING ERROR, 'WHERE INVOKED' REPORT TERMINATED**

**Explanation:** The 'SORT-RETURN' register is greater than zero. The Module Where Invoked Report is not produced.

**Action:** Review possible diagnostics from the sort.

## C.4 SPP Diagnostics for Structured Programming Processor (SPP)

**SPP01E** **STRUCTURE NOT ENDED IN PREV MODULE :** *word*

**Explanation:** The construct designated by "*word*" (IF, SELECT, LOOP, or SEARCH) is incomplete within the preceding module. The corresponding ENDIF, ENDSELECT, ENDLOOP, or ENDSEARCH is required. The condition code is set to 12.

**Action:** Properly end the outstanding constructs and retranslate.

**SPP02E** **'WHEN END' CONDITION MISSPLACED**

**Explanation:** The 'WHEN END' condition was improperly used in a SEARCH construct. If used, the WHEN END must be specified as the first WHEN.

**Action:** Review and correct the syntax of the SEARCH construct.

**SPP07E** **NEST STRUCTURE OVERFLOW**

**Explanation:** 1) SELECT FIRST ACTION FOR identifier constructs have been nested beyond the maximum of nine levels, or 2) over 97 constructs of any kind are nested. The nesting capacity of SPP is exceeded. The condition code is set to 12.

**Action:** Place constructs which are beyond the nesting capacity of SPP in subordinate modules.

**SPP09E** **"GO," "NEXT," OR "THRU" PROHIBITED**

**Explanation:** A COBOL GO TO, NEXT SENTENCE, or PERFORM...THRU has been encountered. All are illegal under SPP. The condition code is set to 12.

**Action:** Replace the GO TO, NEXT SENTENCE, or PERFORM...THRU statement with structured forms. If the statement diagnosed is generated from other CA-MetaCOBOL+ macro sets, translate with or subsequent to SPP translation.

**SPP10E** **STRUCTURE OVERFLOW**

**Explanation:** Over 999 unique constructs are found within a single module. The capacity of SPP is exceeded. The condition code is set to 12.

**Action:** Subdivide the module into a control module with subordinates.

**SPP11E** **INVALID SYNTAX IN "SELECT"**

**Explanation:** Syntax is incorrect following SELECT and prior to first WHEN. The condition code is set to 12.

**Action:** Review the rules for syntax.

**SPP12E** **INVALID SYNTAX FOR "LOOP FOR"**

**Explanation:** The syntax of a LOOP FOR...THRU or LOOP VARYING...THRU construct is illegal. The condition code is set to 12.

**Action:** Review the syntax rules for the LOOP FOR construct and correct.

**SPP13E** **CONSTRUCT INVALID IN ELSE PATH**

**Explanation:** 1) An IF construct contains more than one ELSE, or 2) a SELECT postscript is followed by a WHEN condition. The condition code is set to 12.

**Action:** Review the syntax rules for IF or SELECT construct formation and correct.

**SPP14E** **INVALID SYNTAX IN "SEARCH"**

**Explanation:** Syntax is incorrect following SEARCH and prior to first WHEN. The condition code is set to 12.

**Action:** Review the rules for syntax.

**SPP15E** **STRUCTURE ERROR: CURRENT CONSTRUCT =** *word*

**Explanation:** A syntax error at the construct level, possibly caused by "overlapping" constructs. The "*word*" represents the current outstanding construct (IF, SELECT, SEARCH, LOOP). For example, a WHEN condition statement not within a SEARCH or SELECT construct is a syntax error. The condition code is set to 12.

**Action:** Review the rules for construct "*word*" formation and correct the source.

**SPP16E** **ONLY ONE "WHEN" ALLOWED IN SEARCH ALL**

**Explanation:** Only one "**WHEN** *condition*" clause is allowed with the SEARCH ALL statement (binary search).

**Action:** Use only one "**WHEN** *condition*" clause, or use the SEARCH VARYING construct (serial search).

**SPP17E** **INVALID "***LEAVE/ESCAPE***" IGNORED**

**Explanation:** The emergency termination keyword LEAVE/ESCAPE was found outside the delimiters of a LOOP construct. The condition code is set to 12.

**Action:** Review the rules for LEAVE/ESCAPE and correct.

**SPP18E** **INVALID USE OF START/END DATA**

**Explanation:** START DATA/END DATA encountered in location other than immediately following a module-name header. The condition code is set to 12.

**Action:** Place START DATA immediately following module-name.

**SPP20E** **SECTION MUST BE FOLLOWED BY MODULE**

**Explanation:** A Procedure Division section header is not followed immediately by a module definition. The condition code is set to 12.

**Action:** Review the rules for module and section definition. Include a control module at the beginning of each section.

**SPP21E** **MISSING "END DATA"**

**Explanation:** END DATA is not encountered in a module beginning with START DATA. The condition code is set to 12.

**Action:** Place END DATA after the last local data definition in the module.

**SPP22E** **EXPECTED: "ARE," "SELECTED," VERB, OR CONTROL STRUCTURE; FOUND: "*next module*"**

**SPP23E** **EXPECTED: "ARE," "SELECTED," VERB, OR CONTROL STRUCTURE; FOUND: "*word*"; DELETING, UNTIL NEXT VERB, OR CONTROL STRUCTURE**

**Explanation:** While processing a SELECT postscript, it was determined that the postscript was not followed by the optional words "ARE" or "SELECT," a COBOL verb, or an SP Facility control structure.

**Action:** Examine code and correct statement following the postscript.

**SPP23W** **FLAG UNDEFINED AT THIS POINT**

**Explanation:** Referenced flag-name is not defined at this point during translation.

**Action:** Check for definition of flag.

**SPP24E** **INVALID SYNTAX IN "WHEN"**

**Explanation:** Review rules for conditions allowed in Format 1 of the SELECT. The only allowable condition is a literal. The only allowable compound logical operator is an OR.

**Action:** Review the rules for syntax.

**SPP25W** **MISSING GOBACK, STOP RUN, EXIT PROGRAM IN TOP MODULE**

**Explanation:** No unconditional GOBACK, STOP RUN, or EXIT program was found in the top module. A GOBACK followed by an EXIT-paragraph has been generated to ensure proper program termination.

**Action:** The checking process for logical terminators can be turned off by specifying UPSI5=G.

**SPP26W** **INVALID USE OF** "*TRUE/FALSE*"

**Explanation:** Incorrect syntax has been detected in a "*TRUE*" or "*FALSE*" condition.

**Action:** Review the rules for syntax.

**SPP27E** **INVALID OPERAND IN** "*SET-TRUE/FALSE*"

**Explanation:** The operand found is not a condition-name or a flag-name.

**Action:** Review the SET-TRUE/FALSE syntax for a valid operand name.

**SPP28E** **INVALID SYNTAX IN EXEC CICS STATEMENT**

**Explanation:** No function specified, or END-EXEC missing.

**Action:** Check CICS documentation and change the syntax of the statement accordingly.

**SPP43E** **VARYING "word" IS UNDEFINED**

**Explanation:** The object of the VARYING clause is undefined.

**Action:** Define or correct the VARYING identifier.

## C.5 SSPS Diagnostics

**SPS01E** **DUPLICATE MODULE DEFINITION**

**Explanation:** The current module-name is non-unique in the first 19 characters. The condition code is set to 12.

**Action:** Correct the module-name so that it is unique in the first 19 characters.

**SPS05E** **TOO MANY MODULES**

**Explanation:** The structured source program contains more than 999 modules, and is beyond the capacity of SPS. The current module is not considered for possible stub generation. The condition code is set to 12.

**Action:** Reduce the number of modules, if possible.

## C.6 SSPV Diagnostics for Structured Programming Using VS COBOL II (SP2)

### C.6.1 General

**SPV01E** **ALL AVAILABLE MARKERS HAVE BEEN USED**

**Explanation:** The program is excessively complex, possibly due to use of many ESCAPE statements.

**Action:** Simplify and retranslate the program.

**SPV02E** **SERIOUS ERROR, CONTROL STRUCTURE INDEX IS ZERO**

**Explanation:** The index used to track nested control structures has been reduced to zero, which should not occur.

**Action:** Rerun the translation and obtain a complete listing of the macros and all the input, then contact CA-MetaCOBOL+ Support.

**SPV03W** **FOUND "*word,*" IGNORING, CONTINUING**

**Explanation:** "*Word*" has been determined to be superfluous and has been ignored. This should not effect the translation unless "*word,*" if misspelled, alters the meaning of a statement.

**Action:** Correct the statement as necessary and retranslate the program.

## C.6.2 Module

**SPV04W** **FIRST MODULE IS UNNAMED, WILL USE DEFAULT MODULE NAME FOR EXIT**

**Explanation:** The first module of the Procedure Division does not have a name. A default name (DEFAULT-MODULE-NAME) has been assigned and will be used for the module's exit.

**Action:** User defined.

**SPV05A** **"GOBACK," "STOP RUN" AND "EXIT PROGRAM" MISSING FROM ENTRY MODULE, PROVIDING "GOBACK"**

**Explanation:** The first module of the Procedure Division or an entry module does not end with one of the above terminating statements. A GOBACK statement has been provided.

**Action:** If GOBACK is inappropriate, correct and retranslate the program.

**SPV06W** **WARNING, ENTRY MODULE WILL "FALL THRU" TO NEXT MODULE**

**Explanation:** A UPSI5=G translate-time option has been specified, suspending the reporting of a missing termination statement for an entry module or the first module of the Procedure Division. This message is a warning, indicating that the entry or initial module will "fall through" logically to a subsequent module.

**Action:** If the "falling through" result is inappropriate, add the missing statement and retranslate the program.

## C.6.3 Start/End Data

**SPV07W** **"START DATA" SHOULD IMMEDIATELY FOLLOW THE MODULE NAME**

**Explanation:** Statements have been detected following the module name and preceding the START DATA header.

**Action:** This is improper module structure. Correct the module's structure by moving the statements after the END DATA header.

**SPV08E** **INVALID LEVEL NUMBER (*nn*) FOUND FOLLOWING START DATA. ASSUMING THIS IS A NEW MODULE.**

**Explanation:** Level number "*nn*" is invalid, an END DATA has been omitted, or an item in area A is invalid.

**Action:** Correct and retranslate the program.

**SPV09E** **INVALID LEVEL NUMBER (*nn*) FOUND FOLLOWING START DATA. RESUMING CURRENT PROCEDURE.**

**Explanation:** Level number "*nn*" is invalid or an END DATA has been omitted. Processing resumes with the current module's procedures.

**Action:** Correct and retranslate the program.

**SPV10E** **"END DATA" MUST BE PRECEDED WITH "START DATA"**

**Explanation:** An END DATA has been located which is not preceded by a START DATA.

**Action:** Correct and retranslate the program.

**SPV11E** **"END DATA" MISSING IN PREVIOUS MODULE**

**Explanation:** Either an END DATA has been omitted or a word has been coded in area A that belongs in area B.

**Action:** Correct and retranslate the program.

## C.6.4 Flags Set-True/Set-False

**SPV12E** **FLAG-NAME HAS BEEN OMITTED**

**Explanation:** The flag-name has been omitted from a flag definition.

**Action:** Correct the definition and retranslate the program.

**SPV13W** **THIS FLAG-NAME "*flag-name*" HASN'T BEEN DEFINED YET**

**Explanation:** The flag-name has not been defined, but may be defined later in the program.

**Action:** Move the flag definition to a location preceding all references to the flag-name and retranslate the program.

**SPV14E** **"SET-TRUE" OPERAND(S) INVALID**

**Explanation:** The statement's operands are not flag-names.

**Action:** Correct the statement and retranslate the program.

**SPV15E** "**SET-FALSE**" **OPERAND(S) INVALID**

**Explanation:** The statement's operands are not flag-names.

**Action:** Correct the statement and retranslate the program.

## C.6.5 Control Structures

**SPV16E** **TOO MANY NESTED CONTROL STRUCTURES**

**Explanation:** Too many control structures have been nested (one within another).

**Action:** Simplify the program by placing some of the nested control structures in separate modules and retranslate the program.

**SPV17E** **THERE IS AN UNTERMINATED** "*control-structure*" **IN THE PRECEDING MODULE**

**Explanation:** The specified control structure is lacking a terminator (END?) in the preceding module.

**Action:** Correct and retranslate the program.

**SPV18E** **CODE FOLLOWING** "*verb*" **IS UNREACHABLE**

**Explanation:** Statements coded after GOBACK, STOP RUN, or EXIT PROGRAM cannot be executed.

**Action:** Review the logic, and correct and retranslate the program.

**SPV19E** "*control-structure*" **OPERAND INVALID**

**Explanation:** An operand for the specified control structure is invalid; it may not be defined properly.

**Action:** Correct and retranslate the program.

## C.6.6 IF/ELSE/ENDIF

**SPV20E "IF (NOT) END" MUST FOLLOW "READ" OR "RETURN"**

**Explanation:** The above "standardized condition" must only be coded subsequent to the appropriate verb.

**Action:** Correct and retranslate the program.

**SPV21E "IF (NOT) SIZE ERROR" MUST FOLLOW "ADD," "SUBTRACT," "MULTIPLY," "DIVIDE," OR "COMPUTE"**

**Explanation:** The above "standardized condition" must only be coded subsequent to the appropriate verb.

**Action:** Correct and retranslate the program.

**SPV22E "IF (NOT) INVALID KEY" MUST FOLLOW "READ," "WRITE," "REWRITE," "DELETE," OR "START"**

**Explanation:** The above "standardized condition" must only be coded subsequent to the appropriate verb.

**Action:** Correct and retranslate the program.

**SPV23E "IF (NOT) OVERFLOW" MUST FOLLOW "CALL," "STRING," OR "UNSTRING"**

**Explanation:** The above "standardized condition" must only be coded subsequent to the appropriate verb.

**Action:** Correct and retranslate the program.

**SPV24E "IF (NOT) END-OF-PAGE" MUST FOLLOW "WRITE"**

**Explanation:** The above "standardized condition" must only be coded subsequent to the appropriate verb.

**Action:** Correct and retranslate the program.

**SPV25E "IF (NOT) EOP" MUST FOLLOW "WRITE"**

**Explanation:** The above "standardized condition" must only be coded subsequent to the appropriate verb.

**Action:** Correct and retranslate the program.

**SPV26E** **"ELSE" IS NOT PAIRED WITH AN "IF"**

**Explanation:** An ELSE has been located which is not preceded by an IF.

**Action:** Correct and retranslate the program.

**SPV27E** **"ENDIF" IS NOT PAIRED WITH AN "IF"**

**Explanation:** An ENDIF has been located which is not preceded by an IF.

**Action:** Correct and retranslate the program.

## C.6.7 SELECT/ENDSELECT and SEARCH/ENDSEARCH

**SPV28W** **"ACTION" SHOULD BE "ACTIONS"**

**Explanation:** ACTIONS has been misspelled.

**Action:** Correct and retranslate the program.

**SPV29E** **"AND" INVALID WITH "SELECT FOR"**

**Explanation:** AND is logically improper.

**Action:** Correct and retranslate the program.

**SPV30E** **"WHEN" IS NOT PAIRED WITH A "SELECT" OR "SEARCH"**

**Explanation:** A WHEN has been located which is not preceded by a SELECT or SEARCH.

**Action:** Correct and retranslate the program.

**SPV31E** **"WHEN NONE" IS NOT PAIRED WITH A "SELECT"**

**Explanation:** A "post-script" has been located which is not preceded by a SELECT.

**Action:** Correct and retranslate the program.

**SPV32E** **"WHEN ANY" IS NOT PAIRED WITH A "SELECT"**

**Explanation:** A "post-script" has been located which is not preceded by a SELECT.

**Action:** Correct and retranslate the program.

**SPV33E** **"WHEN SOME" IS NOT VALID WITH "SELECT FIRST"**

**Explanation:** This "post-script" cannot be used with this type of SELECT.

**Action:** Correct and retranslate the program.

**SPV34E** **"WHEN SOME" IS NOT PAIRED WITH A "SELECT"**

**Explanation:** A "post-script" has been located which is not preceded by a SELECT.

**Action:** Correct and retranslate the program.

**SPV35E** **"WHEN NOT ALL" IS NOT VALID WITH "SELECT FIRST"**

**Explanation:** This "post-script" cannot be used with this type of SELECT.

**Action:** Correct and retranslate the program.

**SPV36E** **"WHEN NOT ALL" IS NOT PAIRED WITH A "SELECT"**

**Explanation:** A "post-script" has been located which is not preceded by a SELECT.

Action: Correct and retranslate the program.

**SPV38E** **"WHEN ALL" IS NOT PAIRED WITH A "SELECT"**

**Explanation:** A "post-script" has been located which is not preceded by a SELECT.

**Action:** Correct and retranslate the program.

**SPV39E** **"ENDSELECT" IS NOT PAIRED WITH A "SELECT"**

**Explanation:** An ENDSELECT has been located which is not preceded by a SELECT.

**Action:** Correct and retranslate the program.

**SPV40E** **"ENDSEARCH" IS NOT PAIRED WITH A "SEARCH"**

**Explanation:** An ENDSEARCH has been located which is not preceded by a SELECT.

**Action:** Correct and retranslate the program.

## C.6.8 LOOP/ENDLOOP

**SPV41E** **"ESCAPE" IS FOLLOWED BY** "*verb*"

**Explanation:** ESCAPE is effectively a termination for a sequence of statements or control structures and should not be succeeded by an imperative statement.

**Action:** Review the logic, and correct and retranslate the program.

**SPV42E** **"ESCAPE" NOT WITHIN "LOOP"**

**Explanation:** An ESCAPE has been located which is not enclosed by a LOOP.

**Action:** Correct and retranslate the program.

**SPV43E** **"VARYING** *data-name***," IS UNDEFINED**

**Explanation:** The "*data-name*" indicated is undefined in this program.

**Action:** Correct and retranslate the program.

**SPV44E** **"FROM** *data-name/literal*" **IS INVALID/UNDEFINED**

**Explanation:** The "*data-name*" or literal as specified is invalid or undefined in this context.

**Action:** Review the data-name's definition or correct the literal and retranslate the program.

**SPV45E** "**BY** *data-name/literal" IS INVALID/UNDEFINED*

**Explanation:** The "*data-name*" or literal as specified is invalid or undefined in this context.

**Action:** Review the data-name's definition or correct the literal and retranslate the program.

**SPV46E** "**THRU** *data-name/literal*" **IS UNDEFINED/UNDEFINED**

**Explanation:** The "*data-name*" or literal as specified is invalid or undefined in this context.

**Action:** Review the data-name's definition or correct the literal and retranslate the program.

**SPV47E** "*data-name/literal* **TIMES,"** **IS INVALID**

**Explanation:** The "*data-name*" or *literal* as specified is invalid in this context.

**Action:** Review the data-name's definition or correct the literal and retranslate the program.

**SPV48E** "*keyword*" **IS NOT PAIRED WITH A "LOOP"**

**Explanation:** The indicated keyword (LEAVE, WHILE, or UNTIL) must be enclosed within a LOOP control structure.

**Action:** Correct and retranslate the program.

**SPV49E** **THIS "LOOP" IS ENDLESS**

**Explanation:** The preceding LOOP control structure does not include a termination or ESCAPE.

**Action:** Correct and retranslate the program.

**SPV50E** **"ENDLOOP" IS NOT PAIRED WITH "LOOP"**

**Explanation:** An ENDLOOP which is not preceded by a LOOP has been located.

**Action:** Correct and retranslate the program.

# Index