## COMPILER CONTROL STATEMENTS

The compiler control statements control the compilation process for a particular program and provide such information as the name to be given to the program, the type of processor, the memory size of the NCR Century System for which the program is to be compiled, the type of program listing desired, etc.

Two compiler specification worksheets (also called compiler control sheets) aid the programmer in preparing the compiler control statements. Sheet 1 is always required for a compilation. Sheet 2 is optional; its use depends on the type of options desired by the programmer.

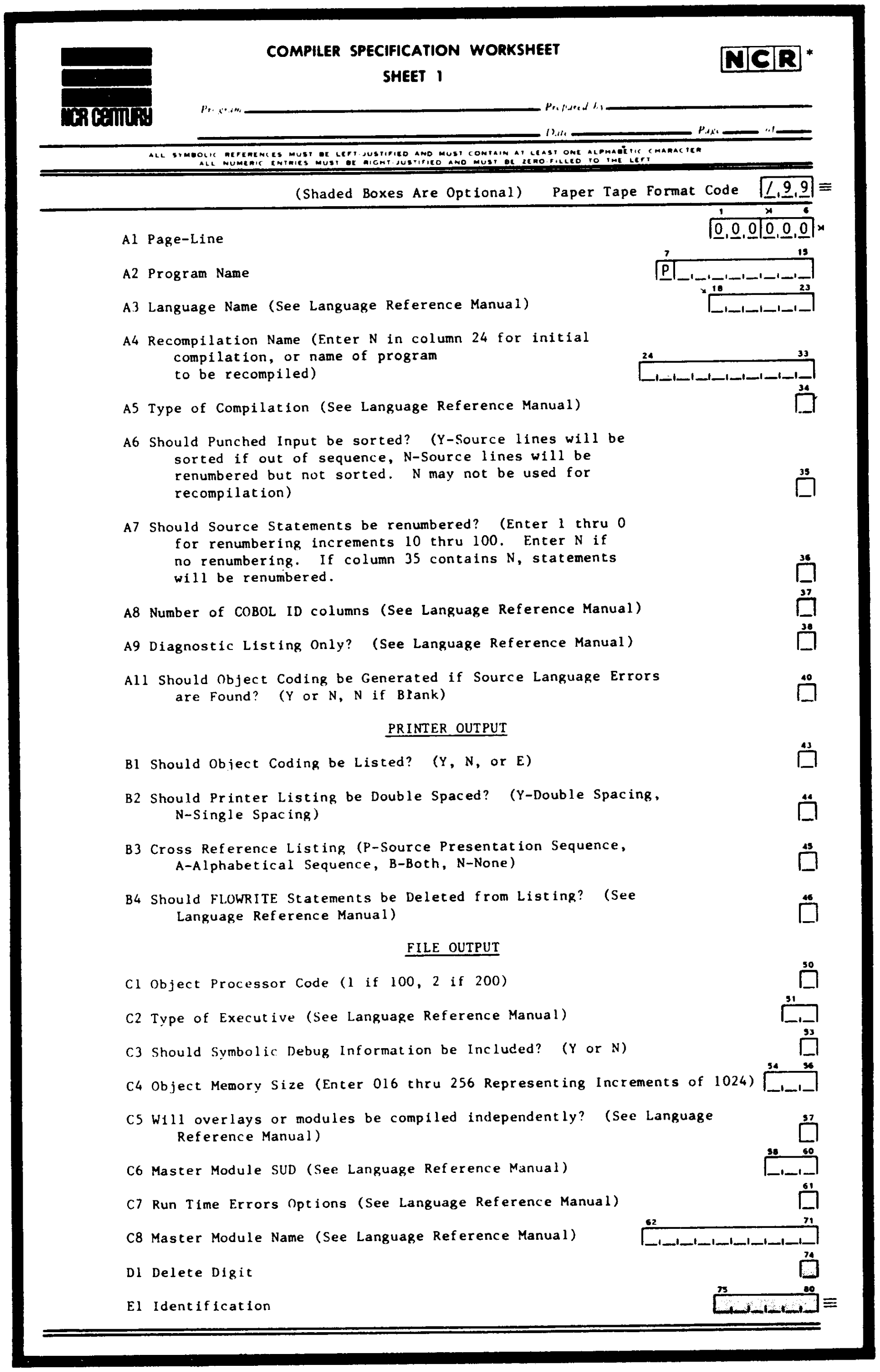
NOTE

> When present, the SUDOP instruction is input between sheets 1 and 2. SUDOP is a SPUR instruction and is not compiled as part of the source program, (see UTILITY ROUTINES MANUAL, PROGRAM ASSOCIATED, Source Program Utility Routines - SPUR").

The programmer's entries on the compiler specification worksheets become the compiler control statements. The compiler control statements must always be the first statements in a source program, as explained in the NEAT/3 REFERENCE MANUAL, COMPILATION PROCESS, tab 1, "Organization of the Source Program."

The following pages show the compiler specifications worksheets and give a detailed description of each worksheet entry.

SHEET 1



COMPILER SPECIFICATION WORKSHEET

SHEET 1

NCR CENTURY NCR*

Program ____ Prepared by ____

____ Date ____ Page ____ of ____

ALL SYMBOLIC REFERENCES MUST BE LEFT-JUSTIFIED AND MUST CONTAIN AT LEAST ONE ALPHABETIC CHARACTER
ALL NUMERIC ENTRIES MUST BE RIGHT-JUSTIFIED AND MUST BE ZERO-FILLED TO THE LEFT

(Shaded Boxes Are Optional) Paper Tape Format Code /99 ≡

| | |
|---|---|
| A1 Page-Line | 1–6: 000000 * |
| A2 Program Name | 7–15: P |
| A3 Language Name (See Language Reference Manual) | 18–23 |
| A4 Recompilation Name (Enter N in column 24 for initial compilation, or name of program to be recompiled) | 24–33 |
| A5 Type of Compilation (See Language Reference Manual) | 34 |
| A6 Should Punched Input be sorted? (Y-Source lines will be sorted if out of sequence, N-Source lines will be renumbered but not sorted. N may not be used for recompilation) | 35 |
| A7 Should Source Statements be renumbered? (Enter 1 thru 0 for renumbering increments 10 thru 100. Enter N if no renumbering. If column 35 contains N, statements will be renumbered. | 36 |
| A8 Number of COBOL ID columns (See Language Reference Manual) | 37 |
| A9 Diagnostic Listing Only? (See Language Reference Manual) | 38 |
| A11 Should Object Coding be Generated if Source Language Errors are Found? (Y or N, N if Blank) | 40 |
| PRINTER OUTPUT | |
| B1 Should Object Coding be Listed? (Y, N, or E) | 43 |
| B2 Should Printer Listing be Double Spaced? (Y-Double Spacing, N-Single Spacing) | 44 |
| B3 Cross Reference Listing (P-Source Presentation Sequence, A-Alphabetical Sequence, B-Both, N-None) | 45 |
| B4 Should FLOWRITE Statements be Deleted from Listing? (See Language Reference Manual) | 46 |
| FILE OUTPUT | |
| C1 Object Processor Code (1 if 100, 2 if 200) | 50 |
| C2 Type of Executive (See Language Reference Manual) | 51 |
| C3 Should Symbolic Debug Information be Included? (Y or N) | 53 |
| C4 Object Memory Size (Enter 016 thru 256 Representing Increments of 1024) | 54–56 |
| C5 Will overlays or modules be compiled independently? (See Language Reference Manual) | 57 |
| C6 Master Module SUD (See Language Reference Manual) | 58–60 |
| C7 Run Time Errors Options (See Language Reference Manual) | 61 |
| C8 Master Module Name (See Language Reference Manual) | 62–71 |
| D1 Delete Digit | 74 |
| E1 Identification | 75–80 ≡ |

The programmer should fill in the worksheet header as defined in the NEAT/3 REFERENCE MANUAL, INTRODUCTION AND DATA, tab 3, "Programming Worksheets."

The paper tape code must be punched if paper tape is used for input to the compiler.

In the following explanations, all applicable items, where practical, are filled with a typical entry.

A1 PAGE-LINE

| 1 |  | 6 |
|---|---|---|
| 000 | 000 | |

The page-line number for the first compiler control sheet must always be 000000 as preprinted in positions one through six.

A2 PROGRAM NAME

| 7 | | 15 |
|---|---|---|
| P | PAYROLL | |

The P in position seven is preprinted and must be punched.

Enter the name of the program being compiled, beginning in position eight. The program name must contain at least one alphabetic character; however, the compiler does not accept the single letter N in position eight as a program name, since this letter has special significance in the recompilation name (see item A4).

NOTE

During an initial compilation, SPUR automatically assigns the version number 00 to the program, except as noted under item A4.

During recompilation, if the program name is the same as the recompilation name, the compiler automatically increments the version number of the program being recompiled by one, and assigns the new version number to the recompiled program.

If the version number of the program being recompiled is 99, enter a new (different) program name in positions eight through 15. The recompiled program thus gets a new program name with a version number of 00. Renaming the program in this way is necessary because a current version number of 99 would be incremented to 00 in the recompiled program, and the highest version number would no longer indicate the newest version of the program.

Refer to the examples of program name entries and recompilation name entries under item A4.

A3 LANGUAGE NAME (SEE LANGUAGE REFERENCE MANUAL)

| 18 | 23 |
|---|---|
| NEAT/3 | |

Enter NEAT/3 in positions 18 through 23 to identify the programming language of the source program.

A4 RECOMPILATION NAME (ENTER N IN COLUMN 24 FOR INITIAL COMPILATION, OR NAME OF PROGRAM TO BE RECOMPILED)

| 24 | 33 |
|---|---|
| N | |

If an initial compilation is being performed, enter the letter N in position 24 and leave positions 25 through 33 blank.

NOTE

> During an initial compilation, if the programmer wishes to assign a version number other than 00 to the program, he must enter the letter N in position 24 and the desired version number, minus one, in positions 32 and 33. During compilation, SPUR increments the number in positions 32 and 33 by one and assigns this version number to the new program.

If a recompilation is being performed, enter the name and version number of the program to be recompiled. The program name must begin in position 24 and the version number must be entered in positions 32 and 33. The acceptable range for the version number is 00 through 99.

If the recompilation name is the same as the program name, the software permits access only to the latest generation of the program (of which there can be two) that is already on the program disc.

Consider the following examples of possible program name and recompilation name entries and the resulting compiled program name and version number.

| PROGRAM NAME [QUESTION A2] | RECOMPILATION NAME [QUESTION A4] | NEW PROGRAM NAME |
|---|---|---|
| ACCTRECD | N | ACCTRECD00 |
| ACCTRECD | NØØØØØØØ49* | ACCTRECD50* |
| ACCTRECD | ACCTRECD02 | ACCTRECD03** |
| RECDACCT | ACCTRECD05 | RECDACCT00 |

* Initial compilation only (see NOTE above).
** If, prior to the recompilation, the program disc already contained versions 00, 01, 02, and 03, the recompilation results in a new generation of version 03. The previous generation of version 03 is then no longer accessible.

## A5 TYPE OF COMPILATION (SEE LANGUAGE REFERENCE MANUAL)

34
F

The entry in position 34 may specify any one of four types of compilation that are available with the NEAT/3 Compiler. In certain instances, the size of the program to be compiled may affect the type (or combination of types) of compilation that is required. Consider the following illustration:

**RELATIONSHIP BETWEEN NEAT/3 COMPILATIONS**

| | Average-Size Programs Not Exceeding 12,000 Source Statements | Larger Programs With Overlays And/Or Modules Exceeding 12,000 Source Statements |
|---|---|---|
| Initial Compilation | Full Compilation | Full Compilation<br>Overlay Compilation |
| Recompilation Entire Program | Full Compilation | Partial Compilation<br>Overlay Compilation |
| Recompilation Overlays Only | Overlay Compilation | Overlay Compilation |
| Recompilation Modules Only | Module Compilation | Module Compilation |

Enter F to specify (1) the initial compilation of an entire program, (2) the recompilation of an entire program, or (3) the initial compilation of the main program (of a program exceeding 12,000 source statements). The initial compilation of large programs requires a second compilation run to compile the program overlays, (see NEAT/3 REFERENCE MANUAL, COMPILATION PROCESS, tab 2, "Independent Overlay Compilation").

Enter P to specify the recompilation of the main program (of a program exceeding 12,000 source statements.) The recompilation of a large program requires a second compilation run to recompile the program overlays, (see publication referenced above).

Enter O to specify the compilation or recompilation of one or more program overlays. The use of independent overlay compilations is discussed in the publication referenced above.

Enter M to specify a module compilation. The use of module compilations is to be discussed in a future publication.

```
A6  SHOULD PUNCHED INPUT BE SORTED?  (Y-SOURCE LINES WILL BE SORTED IF OUT      35
    OF SEQUENCE, N-SOURCE LINES WILL BE RENUMBERED BUT NOT                      [N]
    SORTED.  N MAY NOT BE USED FOR RECOMPILATION)
```

Enter Y if out-of-sequence source statements are to be sorted by page and line number. A recompilation always requires a Y in position 35.

Enter N if (1) this is an initial compilation, (2) a sort is not desired, and (3) source statements are to be renumbered in their order of presentation (see item A7 on the following page).

A7 SHOULD SOURCE STATEMENTS BE RENUMBERED? (ENTER 1 THRU 0 FOR RENUMBERING INCREMENTS 10 THRU 100. ENTER N IF NO RENUMBERING. IF POSITION 35 CONTAINS N, STATEMENTS WILL BE RENUMBERED) 36 [1]

Enter a 1-character decimal number to specify the renumbering increment desired. An entry of 1, 2, 3, ... 9 and 0 reflect increments of 10, 20, 30, ... 90 and 100 respectively.

Enter N if renumbering is not desired and item A6 contains Y.

A8 NUMBER OF COBOL ID COLUMNS (SEE LANGUAGE REFERENCE MANUAL) 37 [ ]

This entry does not apply to the NEAT/3 language and should be left blank.

A9 DIAGNOSTIC LISTING ONLY? (SEE LANGUAGE REFERENCE MANUAL) 38 [P]

Enter P to specify a precompilation run is to be performed, for which the programmer requests a diagnostic listing only.

A11 SHOULD OBJECT CODING BE GENERATED IF LANGUAGE ERRORS ARE FOUND? (Y OR N, N IF BLANK) 40 [Y]

Enter Y if object coding is to be generated if source language errors are found.

Enter N or leave blank if object coding is not to be generated for source language errors.

B1 SHOULD OBJECT CODING BE LISTED? (Y, N, OR E) 43 [Y]

Enter Y if the object coding created for each source line of the program is to be listed as part of the compiler listing.

Enter E if the extended object coding created for the program is to be listed as part of the compiler listing.

Enter N if no listing of the object coding is desired. If left blank, N is assumed.

B2 SHOULD PRINTER LISTING BE DOUBLE SPACED? (Y-DOUBLE SPACING, N-SINGLE SPACING) 44 [N]

Enter Y to indicate that the compiler listing should be double spaced.

Enter N to indicate the compiler listing should be single-spaced. If left blank, N is assumed.

B3 CROSS REFERENCE LISTING (P-SOURCE PRESENTATION SEQUENCE, A-ALPHABETICAL SEQUENCE, B-BOTH, N-NONE) — 45 $\boxed{N}$

Enter P if the cross reference listing is to be printed in presentation sequence. Each reference is listed with the page and line numbers of all source statements that use that reference in their operands column. The listing is in descending order with the reference in the source statement with the highest page-line number printed first.

Enter A if the cross reference listing is to be printed in alphabetical order. Each reference is listed with the page and line numbers of all source statements that use that reference in their operands column.

Enter B if the cross reference listing is to be printed in both alphabetical order and presentation sequence.

Enter N if the cross reference listing should not be printed. If left blank, N is assumed.

B4 SHOULD FLOWRITE STATEMENTS BE DELETED FROM LISTING? (Y OR N) — 46 $\boxed{Y}$

Enter Y if Flowrite statements are to be deleted from the compiler listing.

Enter N if Flowrite statements are to be included in the compiler listing. If left blank, N is assumed.

C1 OBJECT PROCESSOR CODE (1 IF 100, 2 IF 200) — 50 $\boxed{1}$

This entry indicates the type of processor for which this program is to be compiled.

Enter 1 to specify an NCR Century 100 processor.

Enter 2 to specify an NCR Century 200 processor.

C2 TYPE OF EXECUTIVE (SEE LANGUAGE REFERENCE MANUAL) — 51 $\boxed{0\,0}$

Enter the symbolic identifier of the executive system for which this program is to be compiled.

C3 SHOULD SYMBOLIC DEBUG INFORMATION BE INCLUDED? (Y OR N) — 53 $\boxed{Y}$

Enter Y to indicate that compiler-generated symbolic debug information is to be included in the object program being compiled. The Symbolic Debug System can only function with a program that includes the symbolic debug information.

Enter N if the symbolic debug information is not to be included in the object program. The option to exclude the symbolic debug information from an object program is normally used after a program has been debugged and is ready to be copied to a production disc pack. If left blank, N is assumed.

C4 OBJECT MEMORY SIZE (ENTER 016 THRU 256 REPRESENTING INCREMENTS OF 1024) — 54–56: 0 1 6

Enter a 3-digit decimal number to indicate the memory size of the system on which this program is to run. This number represents increments of 1024. For instance, the number 016 specifies a memory size of 16,384 characters.

If the programmer does not specify the object memory size, or if positions 54 through 56 contain an invalid entry, the compiler automatically assumes that the object memory size is the same as the memory size of the system on which the program is to be compiled.

C5 WILL OVERLAYS OR MODULES BE COMPILED INDEPENDENTLY? (SEE LANGUAGE REFERENCE MANUAL) — 57: Y

Enter Y to indicate that overlays or modules will be compiled independently from the main program.

Enter N to indicate that overlays will not be compiled independently. If left blank, N is assumed.

C6 MASTER MODULE SUD (SEE LANGUAGE REFERENCE MANUAL) — 58–60

If a module compilation has been specified (M) in item A5, enter the symbolic unit designator of the peripheral on which the object master module is mounted. For other types of compilation, this entry may be left blank. (See TECHNIQUES AND PROCEDURES MANUAL, COMPILER RELATED, "Object Module Assembler Program - OMAP.")

C7 RUN TIME ERROR OPTIONS (SEE LANGUAGE REFERENCE MANUAL) — 61

This entry does not apply to the NEAT/3 language and should be left blank.

C8 MASTER MODULE NAME (SEE LANGUAGE REFERENCE MANUAL) — 62–71

If a module compilation has been specified (M) in item A5, enter the name of object master module that is contained on the unit designated in item C6. For other types of compilation, this entry may be left blank. Refer to the reference for item C6.

D1 DELETE DIGIT — 74: N

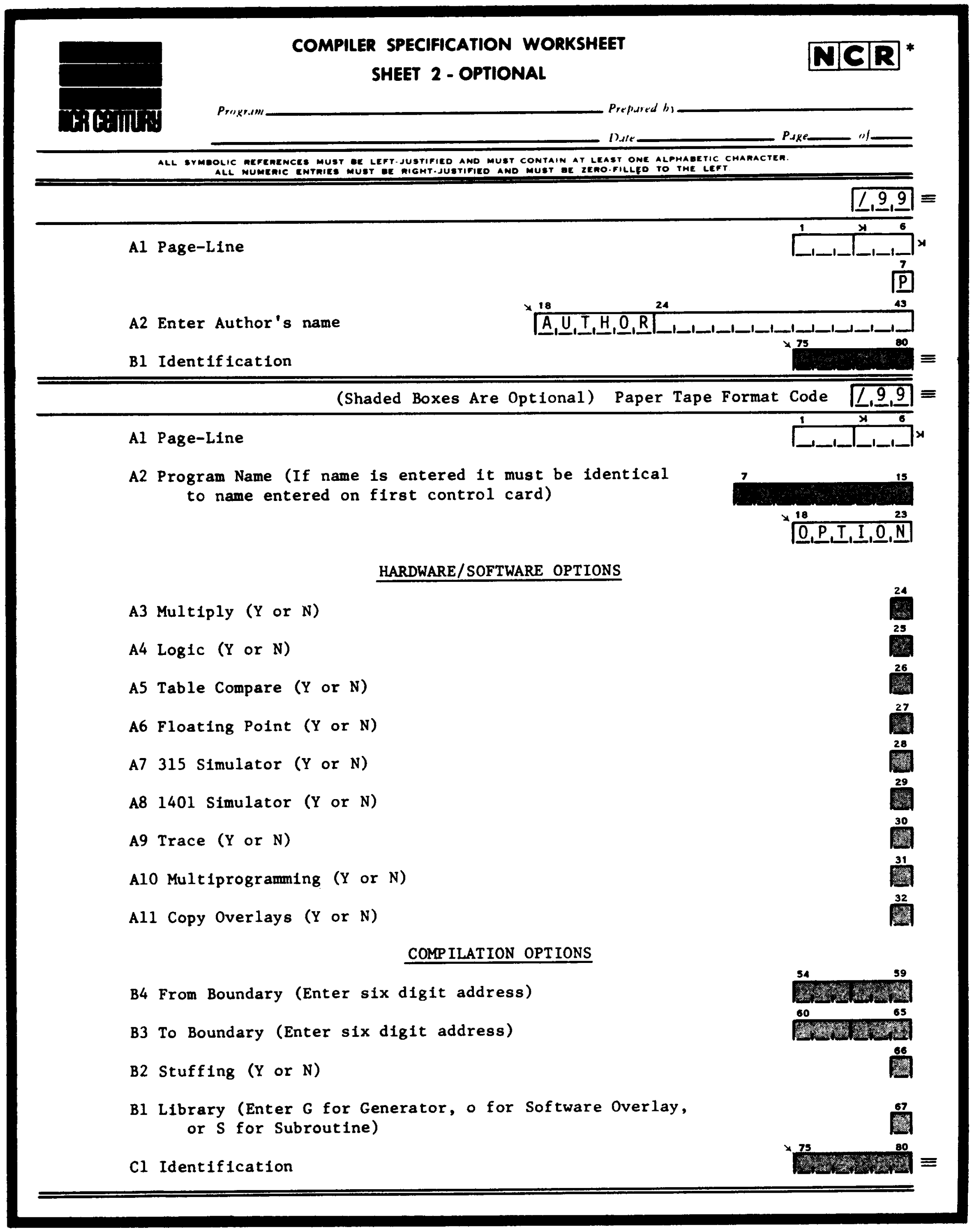
This entry controls the elimination of specific source statements in the program being compiled. The compilation process ignores all source statements that have a delete digit of equal or lower value than the entry made in position 74 of the compiler control statement. For instance, the numeral 3 in position 74 causes all source statements with a delete digit of 3, 2 or 1 to be dropped from the program. Source statements with no entry in the delete digit position can never be dropped.

Enter N if the delete digit option is not to be used.

E1 IDENTIFICATION — 75–80

Enter the program identification as explained in the NEAT/3 REFERENCE MANUAL, INTRODUCTION AND DATA, tab 3, "Programming Worksheets."

SHEET 2



NCR CENTURY

**COMPILER SPECIFICATION WORKSHEET**

**SHEET 2 - OPTIONAL**

NCR*

Program ________ Prepared by ________

________ Date ________ Page ____ of ____

ALL SYMBOLIC REFERENCES MUST BE LEFT-JUSTIFIED AND MUST CONTAIN AT LEAST ONE ALPHABETIC CHARACTER.
ALL NUMERIC ENTRIES MUST BE RIGHT-JUSTIFIED AND MUST BE ZERO-FILLED TO THE LEFT

| | | |
|---|---|---|
| | | /99 ≡ |
| A1 Page-Line | 1–6 | |
| | 7 | P |
| A2 Enter Author's name | 18–43 | AUTHOR |
| B1 Identification | 75–80 | ▨ ≡ |

(Shaded Boxes Are Optional) Paper Tape Format Code /99 ≡

| | | |
|---|---|---|
| A1 Page-Line | 1–6 | |
| A2 Program Name (If name is entered it must be identical to name entered on first control card) | 7–15 | ▨ |
| | 18–23 | OPTION |

HARDWARE/SOFTWARE OPTIONS

| | | |
|---|---|---|
| A3 Multiply (Y or N) | 24 | ▨ |
| A4 Logic (Y or N) | 25 | ▨ |
| A5 Table Compare (Y or N) | 26 | ▨ |
| A6 Floating Point (Y or N) | 27 | ▨ |
| A7 315 Simulator (Y or N) | 28 | ▨ |
| A8 1401 Simulator (Y or N) | 29 | ▨ |
| A9 Trace (Y or N) | 30 | ▨ |
| A10 Multiprogramming (Y or N) | 31 | ▨ |
| A11 Copy Overlays (Y or N) | 32 | ▨ |

COMPILATION OPTIONS

| | | |
|---|---|---|
| B4 From Boundary (Enter six digit address) | 54–59 | ▨ |
| B3 To Boundary (Enter six digit address) | 60–65 | ▨ |
| B2 Stuffing (Y or N) | 66 | ▨ |
| B1 Library (Enter G for Generator, o for Software Overlay, or S for Subroutine) | 67 | ▨ |
| C1 Identification | 75–80 | ▨ ≡ |

The programmer should fill in the worksheet header as defined in the NEAT/3 REFERENCE MANUAL, INTRODUCTION AND DATA, tab 3, "Programming Worksheets."

This sheet is used by the programmer in preparing certain optional statements, which may be applicable in the system for which the program is to be compiled.

## AUTHOR STATEMENT (OPTIONAL)

The author statement is an optional entry, whereby the programmer may include the name of the author of the program being compiled. When specified, the author statement is included in the program listing that is produced by the compilation process.

If punched paper tape is used for input to the compiler, the paper tape format code must be punched before the page and line number for the author's statement.

A1 PAGE-LINE — 1–6: 000001

Enter a six-digit page-line number in positions one through six to indicate the input sequence for the author's statement. Normally this number will be 000001.

NOTE

SUDOP Instruction may precede the author's statement in actual input sequence, however, SUDOP instructions do not require page-line numbers since they are not compiled as part of the source program.

The P in position seven is preprinted and must be punched. — 7: P

A2 ENTER AUTHOR'S NAME — 18: AUTHOR 24: GEORGE WANGTON 43

The entry AUTHOR in positions 18 through 23 is preprinted and must be punched.

Enter the author's name beginning in position 24. Normally the author's name will be the same as the programmer's name. When requested, the author's name is printed in the header of the program listing.

B1 IDENTIFICATION — 75–80

Enter the program identification as explained in the NEAT/3 REFERENCE MANUAL, INTRODUCTION AND DATA, tab 3, "Programming Worksheets."

The second part of this sheet is used to specify certain options that may be available in the system for which this program is being compiled. The listed features, which are not available on NCR Century 100 Systems, are intended only for higher members of the NCR Century Series.

If punched paper tape is used for input, the paper tape format code must be punched before the page and line number for the options statement.

A1 PAGE-LINE — positions 1–6: 000002

Enter a six-digit page-line number in positions one through six to indicate the input sequence for the options statement. Normally this number will be 000002.

A2 PROGRAM NAME (IF NAME IS ENTERED IT MUST BE IDENTICAL TO NAME ENTERED ON FIRST CONTROL CARD) — positions 7–15: P

The P in position seven is preprinted and must be punched. — positions 18–23: OPTION

The program name is an optional entry. If the programmer makes an entry in positions eight through 15, it must be identical to the program name specified on Sheet 1.

The entry OPTION in positions 18 through 23 is preprinted and must be punched.

## HARDWARE/SOFTWARE OPTIONS

| | | |
|---|---|---|
| A3 | MULTIPLY (Y OR N) | 24 |
| A4 | LOGIC (Y OR N) | 25 |
| A5 | TABLE COMPARE (Y OR N) | 26 |
| A6 | FLOATING POINT (Y OR N) | 27 |
| A7 | 315 SIMULATOR (Y OR N) | 28 |
| A8 | 1401 SIMULATOR (Y OR N) | 29 |
| A9 | TRACE (Y OR N) | 30 |
| A10 | MULTIPROGRAMMING (Y OR N) | 31 |

Items A3 through A10 are optional entries. If the programmer enters Y for any of the above listed optional hardware features, the compiler will generate an object program capable of using the specified feature(s).

Enter N or leave blank, if any feature listed above is either not available, or is not to be used.

A11 COPY OVERLAYS (Y OR N) — position 32

This entry is only for the use of software programmers. Generally position 32 should be left blank.

## COMPILATION OPTIONS

B4 FROM BOUNDARY (ENTER SIX DIGIT ADDRESS) — positions 54–59

Enter six-digit address in memory, at which this program is to begin.

60 65

B3 TO BOUNDARY (ENTER SIX DIGIT ADDRESS)

Enter six-digit address in memory, at which this program is to end.

66

B2 STUFFING (Y OR N)

Enter Y if stuffing is desired.

Enter N or leave blank if stuffing is not desired.

67

B1 LIBRARY (ENTER G FOR GENERATOR, O FOR SOFTWARE OVERLAY, OR S FOR SUBROUTINE)

If the program being compiled is to be a library entry, enter the appropriate letter to indicate the category.

75 80

C1 IDENTIFICATION

## COPY CONTROL INSTRUCTIONS

The NEAT/3 language includes three versions of the COPY control instruction: COPYA, COPYP, and COPYR.

### COPYA

#### Function

The COPYA instruction specifies an existing source program and inserts it into the source program of which the COPYA instruction is a part. The COPYA instruction copies the entire specified program except the compiler control statement.

The COPYA instruction automatically assigns new page and line numbers to the copied source statements. The first source statement to be copied receives a page and line number that is ten greater than the page and line number of the COPYA instruction itself. The page and line numbers of subsequent source lines to be copied are again incremented by ten each.

#### Example

|  | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C |  | COPYA | PROGRAM001 |

The above COPYA instruction inserts source program PROGRAM001 (minus compiler control statements) in the source program of which the above instruction is a part. The first source statement of PROGRAM001 will be assigned page and line number 005040.

#### Conventions

The program specified in the COPYA instruction must be available on the currently mounted program disc.

The programmer must provide a sufficiently large page and line number gap between the COPYA instruction and the following coded instruction to allow for the insertion of the copied source program.

COPYA instructions may be coded on either coding sheets or data layout sheets, depending upon the format of the first statement to be copied. If a data layout sheet is used, the operands must begin in position 31.

## COPYP

### Function

The COPYP instruction specifies an existing source program from which a range of contiguous source statements is to be copied and inserted into the program of which the COPYP instruction is a part.

The COPYP instruction automatically assigns new page and line numbers to the copied source statements. The first source statement to be copied receives a page and line number that is ten greater than the page and line number of the COPYP instruction itself. The page and line numbers of subsequent source lines to be copied are again incremented by ten each.

### Example

|  | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C |  | COPYP | PROGRAM003,031030,033090 |

The first entry in the OPERANDS column (PROGRAM003) is the name of the program from which source statements are to be copied.

The second entry in the OPERANDS column (031030) is the page and line number of the first source statement to be copied. If this second entry specifies a page and line number that cannot be found in PROGRAM003, the copy process starts with the source statement that has the next higher page and line number.

The third entry in the OPERANDS column (033090) is the page and line number of the first source statement following the source statements to be copied; i.e., the copy process stops after copying the source statement immediately preceding the one indicated in the third entry.

### Conventions

The program specified in the COPYP instruction must be available on the currently mounted program disc.

The programmer must provide a sufficiently large page-and-line number gap between the COPYP instruction and the following coded instruction to allow for the insertion of the copied source statements.

To copy to the end of a program, the programmer must use END$ as the third entry in the OPERANDS column of the COPYP instruction.

COPYP instructions may be coded on either coding sheets or data layout sheets, depending upon the format of the first statement to be copied. If a data layout sheet is used, the operands must begin in position 31.

## COPYR

### Function

The COPYR instruction has the same function as the COPYP instruction explained on the preceding page. However, COPYR uses reference names to specify a program range where COPYP uses page-and-line numbers.

### Example

| | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | | COPYR | PROGRAM059,ABLE,BAKER |

The first entry in the OPERANDS column (PROGRAM059) is the name of the program from which source statements are to be copied.

The second entry in the OPERANDS column (ABLE) is the reference of the first source statement to be copied.

The third entry in the OPERANDS column (BAKER) is the reference of the first source statement following the source statements to be copied; i.e., the copy process stops after copying the source statement immediately preceding the one indicated in the third entry.

### Conventions

The program specified in the COPYR instruction must be available on the currently mounted program disc.

To copy to the end of a program, the programmer must use END$ as the third entry in the OPERANDS column of the COPYP instruction.

COPYR instructions may be coded on either coding sheets or data layout sheets, depending upon the format of the first statement to be copied. If a data layout sheet is used, the operands must begin in position 31.

The reference of a COPYR instruction may be qualified; however, only one qualifier may be used.

## OMIT CONTROL INSTRUCTION

### OMIT

#### Function

To debug or alter a program, the programmer must usually delete, replace, and/or insert some source statements during a recompilation. During the recompilation, the OMIT instruction prevents a source line or group of source lines from being copied from the old recompilation master to the new recompilation master being generated.

The OMIT instruction is also useful during initial compilation, especially if paper tape is used as input media. To cancel erroneous source statements or groups of source statements from the input media, OMIT instructions may be input on the same input media.

#### Example of Omitting a Single Source Statement

| PAGE | LINE | | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|---|---|
| 1 2 3 | 4 5 6 | 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| 038 | 090 | C | | OMIT | |

The programmer omits an individual source statement by assigning to an OMIT instruction the page and line number of the source statement to be omitted.

The above OMIT instruction omits the source statement with page and line number 038090 from a recompilation master or from the source program of which it is a part.

Example of Omitting a Group of Source Statements

| PAGE | LINE | | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|---|---|
| 1 2 3 | 4 5 6 | 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| 048 | 150 | C | | OMIT | 049030 |

To omit a group of source statements, the programmer specifies a range of page-and-line numbers in his OMIT instruction. The page and line number of the OMIT instruction itself is the page and line number of the first source statement to be omitted; the page and line number entered in the OPERANDS column of the OMIT instruction indicates the page-and-line number of the last source line to be omitted.

The above OMIT instruction omits the source statements whose page-and-line numbers range from 048150 to 049030.

Conventions

The OMIT instruction may be coded on either coding sheets or data layout sheets, depending upon the format of the first statement to be omitted.

During the recompilation process, the recompilation master must be available on one of the system disc packs, and the programmer must enter the OMIT statements through the system card reader or the system paper tape reader. The programmer specifies the name of the recompilation master on the compiler control statement preceding the OMIT instruction.

Whenever the OMIT instruction is used as part of an initial source program, the compiler control statement should specify sorting of the source statements by page-and-line numbers.

## OVERLAY CONTROL INSTRUCTIONS

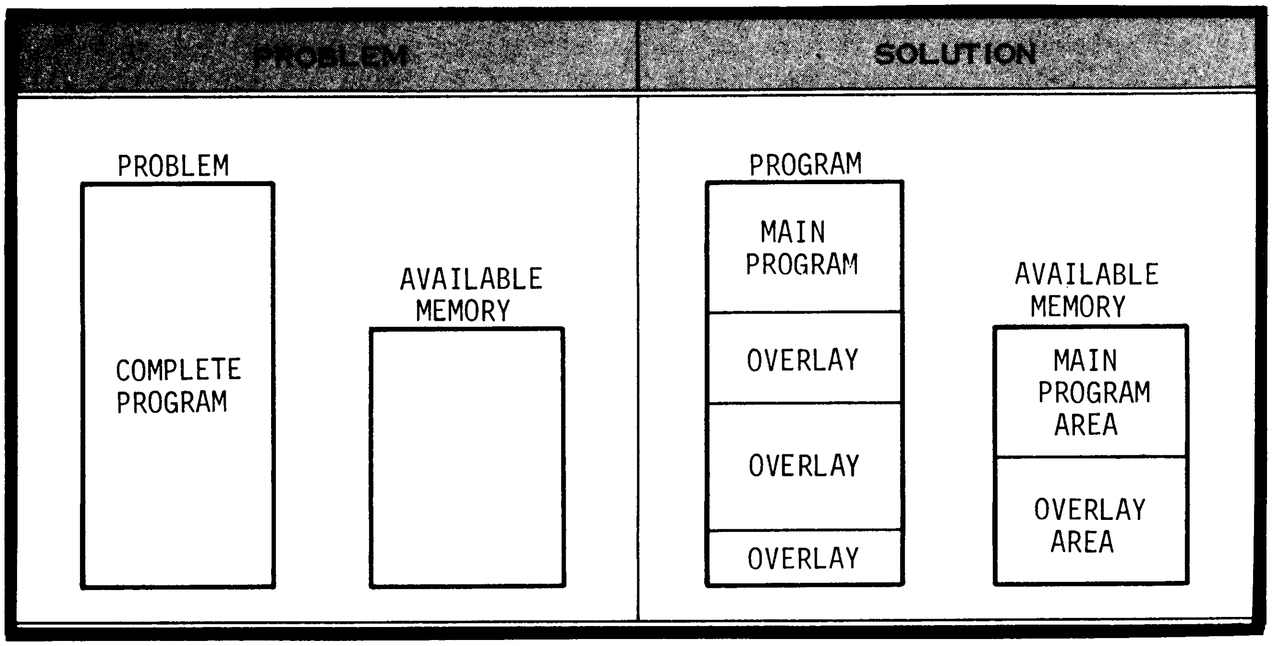
The NEAT/3 language provides two overlay control instructions, OVRLAY and OVRLAYG.

### PROGRAM OVERLAY CONCEPTS

The concept of program overlays provides the programmer with the ability to divide a program into two or more logical segments, permitting greater flexibility during both the compilation process and the actual program run.

- The primary application for the use of program overlays is designed for those programs that are too large to fit into the memory area available for the user's programs. By designating these logical segments as program overlays, the program may then be processed in smaller units that are compatible with the available memory area. Normally, the entire program, including overlays, is compiled in a single run, which makes the overlays readily accessible on disc. During the program run, overlays are automatically called into the overlay area as needed by the program. As each new overlay is called into the overlay area, it replaces the previous overlay.

  Since program overlays represent logical segments of a complete program, the overlays need not be of equal length. During the compilation process, the NEAT/3 Compiler automatically reserves an overlay area in memory that is of sufficient size to accommodate the largest overlay in the program being compiled. Consider the following illustration.



- In addition, the use of program overlays provides the programmer with the ability to compile overlays independently from the main program. This ability permits several specific applications: (1) minor changes to one or more overlays may be made without recompiling the entire program; (2) a program may be compiled in which the programmer has used the OVRLAY control instructions to define overlays that are to be added at a later date; and (3) programs exceeding 12,000 source statements may be compiled. Independent overlay compilation is discussed in a separate publication, (see NEAT/3 REFERENCE MANUAL, COMPILATION PROCESS, tab 2, "Independent Overlay Compilation").

The basic function of the OVRLAY and OVRLAYG instructions is the same for all of the previously stated applications; that is, to identify the start of an overlay or to identify the start of an overlay group and the first overlay within the group. The functions of each of these instructions are discussed under separate headings in this publication. Additional functions, which are applicable only to independent overlay compilation, are either noted as such, or direct the reader to the above referenced publication.

## OVRLAY

### Function

The OVRLAY instruction is used as the overlay header, which (1) identifies the start of a program overlay, and (2) informs the NEAT/3 Compiler to accept all subsequent data and coding as part of that overlay. A new OVRLAY instruction is required for each overlay in a program.

The first OVRLAY instruction in a program also identifies the start of an overlay group. All subsequent overlay groups must be started with an OVRLAYG instruction. The grouping of overlays is discussed in more detail under the OVRLAYG instruction in this publication.

In addition, the OVRLAY instruction automatically starts a new program section, by performing the same functions as the SECT control instruction, (see NEAT/3 REFERENCE MANUAL, INSTRUCTIONS, tab 3, "Section Control Instruction").

NOTE

The SECT control instruction may be used within an overlay to divide the overlay into sections.

### Example

The OVRLAY instruction may be input to the Compiler as either a data statement or a coding statement. The use of each is as follows:

- Data Statement

  If an overlay contains data statements in addition to the program coding, the OVRLAY instruction must be coded on a data layout sheet, since the input sequence to the NEAT/3 Compiler requires that data statements precede coding statements.

  If overlays are to be compiled independently and it is anticipated that data statements may be added to an overlay at a later date, the OVRLAY instruction must also be coded on a data layout sheet to accommodate for the input sequence to the Compiler.

| ✕ | REFERENCE | ✕ CODE | LOCATION | ✕ LENGTH | ✕ DP | ✕ TYPE | VALUE OR PICTURE |
|---|---|---|---|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 | 24 25 26 27 | 28 29 | 30 | 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| D | DEDUCTIONS | O | VRLAY | | | | |

The OVRLAY instruction may contain a reference tag in positions 8-17 to serve as a qualifier for entry points into the overlay, (see NEAT/3 REFERENCE MANUAL, INSTRUCTIONS, tab 3, "ENTRY Control Instruction").

● Coding Statement

If an overlay contains only coding statements, and the addition of data statements is not anticipated, the OVRLAY instruction may be coded on a coding sheet.

|  | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | D E D U C T I O N S | O V R L A Y |  |

In both of the preceding illustrations, the OVRLAY instruction is used to begin a new overlay, which specifies the qualifier DEDUCTIONS for all references within the overlay.

The relationship between the internal structure of an overlay and the use of data and coding statements for coding the OVRLAY instruction is shown in the following illustration. The type of statement (D for data, C for coding) is indicated in parentheses.

**OVERLAY STRUCTURES**

| A | B | C |
|---|---|---|
| OVRLAY (D) | OVRLAY (D) | OVRLAY (C) |
| ENTRY (D) | ENTRY (D) | ENTRY (C) |
| ENTRY (D) | ENTRY (D) | ENTRY (C) |
| Data (D) | **Data to be** | Coding (C) |
| Data (D) | **added at a** | Coding (C) |
| Data (D) | **later date** | Coding (C) |
| Coding (C) | Coding (C) | Coding (C) |
| Coding (C) | Coding (C) | Coding (C) |

- In example A, the overlay contains its own data definitions, which require the OVRLAY instruction to be coded on a data layout sheet.
- In example B, the overlay is expected to contain its own data definitions. The OVRLAY instruction is coded on a data layout sheet to permit the addition of data statements at a later date.

Another example might be, if the overlay is contained within a previously defined overlay group, the OVRLAY instruction may be used to define a proposed overlay, which remains to be written or completed. The OVRLAY instruction is coded on a data layout sheet to permit the addition of both data and coding statements.

- In example C, the overlay contains only coding statements. The OVRLAY instruction may be coded on a coding sheet.

NOTE

The ENTRY instruction, if present, must always be the same type of statement as that used for the OVRLAY instruction.

## Conventions

If a program specifies branching to an overlay that is not currently in the overlay area, the software inputs the needed overlay. The new overlay replaces the previous overlay contained in the overlay area.

Overlays are always called into memory from disc in their original form, but are never written back onto disc. Therefore, the programmer should not locate any working storage areas in an overlay, unless the contents of these areas are no longer needed once the overlay is replaced by a new overlay.

Th programmer must place all file specifications at the beginning of the main program. Normally, the programmer also places all area definitions in the main program because of the following programming restrictions; (1) instructions in the main program can only access those areas that are defined in the main program, and (2) instructions in an overlay can only access those areas that are defined either in the main program or in the same overlay as the instruction.

The use of a data reference tag as the operand of an instruction cannot call an overlay into memory. Overlays can only be called into memory by branching to an entry point within the overlay.

The following NEAT/3 instructions are capable of calling overlays. "Z" in the OPERANDS column represents the reference of an instruction in the program. If this reference is also specified as an entry point to an overlay (see, NEAT/3 REFERENCE MANUAL, INSTRUCTION, tab 3, "ENTRY Control Instruction"), the instruction containing this reference is capable of calling an overlay. Consider the following illustration.

| INST. | OPERAND | INST. | OPERAND | INST. | OPERAND |
|---|---|---|---|---|---|
| ADDC | A,B,Z | BRL | Z | TBILDN | A,B,Z |
| ADDRC | A,B,Z | BRGE | Z | TBILDD | A,B,C,Z |
| SUBC | A,B,Z | BRLE | Z | TFINDN | A,Z |
| SUBRC | A,B,Z | BRU | Z | TFINDD | A,B,Z |
| MULTC | A,B,C,Z | LINK | Z | TFINDR | A,B,C,Z |
| MULTRC | A,B,C,Z | RELINK | Z | TFINDS | A,B,C,Z |
| DIVC | A,B,C,Z | RGET | A,B,Z | TFINDB | A,B,C,Z |
| DIVRC | A,B,C,Z | SGET | A,Z | TSERT | A,B,Z |
| BR | Z | SGETL | A,Z1,Z2 | TDEL | A,B,Z |
| BRE | Z | SGETC | A,Z1,Z2 | TPACK | A,Z |
| BRG | Z | BLKCHK | A,B,Z | TSET | A,B,Z |

If the RELINK instruction transfers control back to an overlay that is no longer in memory, that overlay is automatically called.

A reference on a file specification sheet (end-of-file or end-of-page) may not specify an entry point to an overlay; any entry point to an overlay can only be specified by an operand of an instruction in the main program.

Any one program is limited to a maximum of 255 overlays.

It is recommended that the OVRLAY instruction normally be coded on a data layout sheet.

## OVRLAYG

### Function

The function of the OVRLAYG instruction is the same as the OVRLAY instruction, which (1) identifies the start of a program overlay, and (2) informs the NEAT/3 Compiler to accept all subsequent data and coding as part of that overlay.

In addition, the OVRLAYG instruction identifies the start of a new overlay group.

When all program overlays are contained in a single group, only one overlay can be called into memory at a time. Each new overlay input to the overlay area replaces the previous overlay.

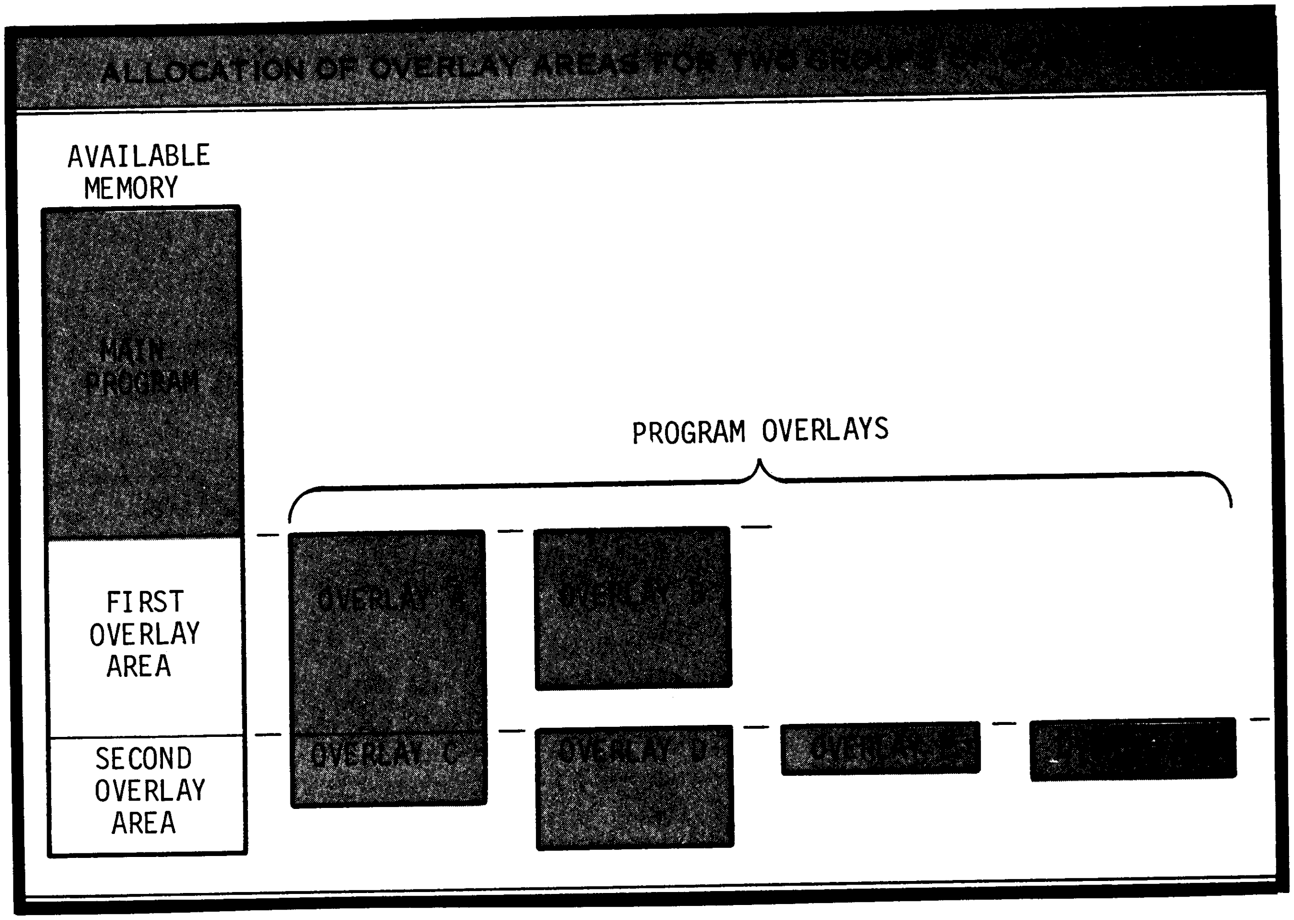
The use of the OVRLAYG instruction permits the programmer to further organize his program overlays into logical groups. The NEAT/3 Compiler assigns a separate memory area to each overlay group. The largest overlay in a group determines the size of the overlay area reserved for that group.

By arranging program overlays into groups, it is possible to have more than one overlay in memory at the same time, i.e., one overlay from each overlay group. For example, when branching from an overlay in one group to an overlay in another group, the overlay in the first group remains accessible in memory. Overlays are only replaced by other overlays from the same group.

The first OVRLAY instruction in a program always starts the first overlay group. Only subsequent overlay groups must be started with an OVRLAYG instruction.

NOTE

> If a program contains two or more overlay groups and overlays are to be compiled independently, then each group must begin with an OVRLAYG instruction and the group size must be defined.



Example

The OVRLAYG instruction may be input to the Compiler as either a data statement or a coding statement. The use of each is as follows:

- Data Statement

  If the first overlay in the group (which the OVRLAYG instruction identifies) contains data statements, the OVRLAYG instruction must be coded on a data layout sheet.

  If overlays are to be compiled independently and it is anticipated that data statements may be added to the overlay at a later date, the OVRLAYG instruction must be coded on a data layout sheet to accommodate for the input sequence to the Compiler.

| X | REFERENCE | X CODE | LOCATION | X LENGTH | X DP | X TYPE | VALUE OR PICTURE |
|---|---|---|---|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 | 24 25 26 27 | 28 29 | 30 | 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| D | O V E R T I M E | O | V R L A Y | | | | G , 5 1 2 |

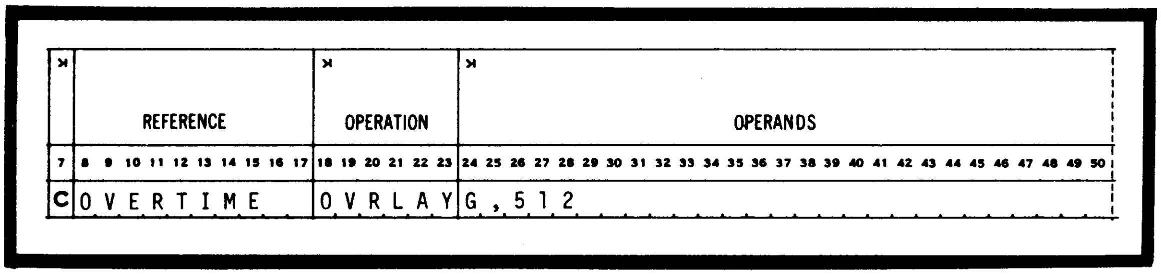
The OVRLAYG instruction may contain a reference tag in positions 8-17 to serve as a qualifier for entry points into the overlay. The reference tag does not apply to the overlay group.

The letter G is entered as the first operand in position 31 to identify the beginning of a new group.

A group size may be specified as the second operand beginning in position 33. A group size is required for all partial and independent overlay compilations.

- <u>Coding Statements</u>

If the overlay contains only coding statements, into which data statements are not to be added, the OVRLAYG instruction may be coded on a coding sheet.



|  | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | OVERTIME | OVRLAY | G,512 |

A reference tag for the overlay may be entered in positions 8-17.

The letter G is entered as the first operand in position 24 to identify the beginning of a new group.

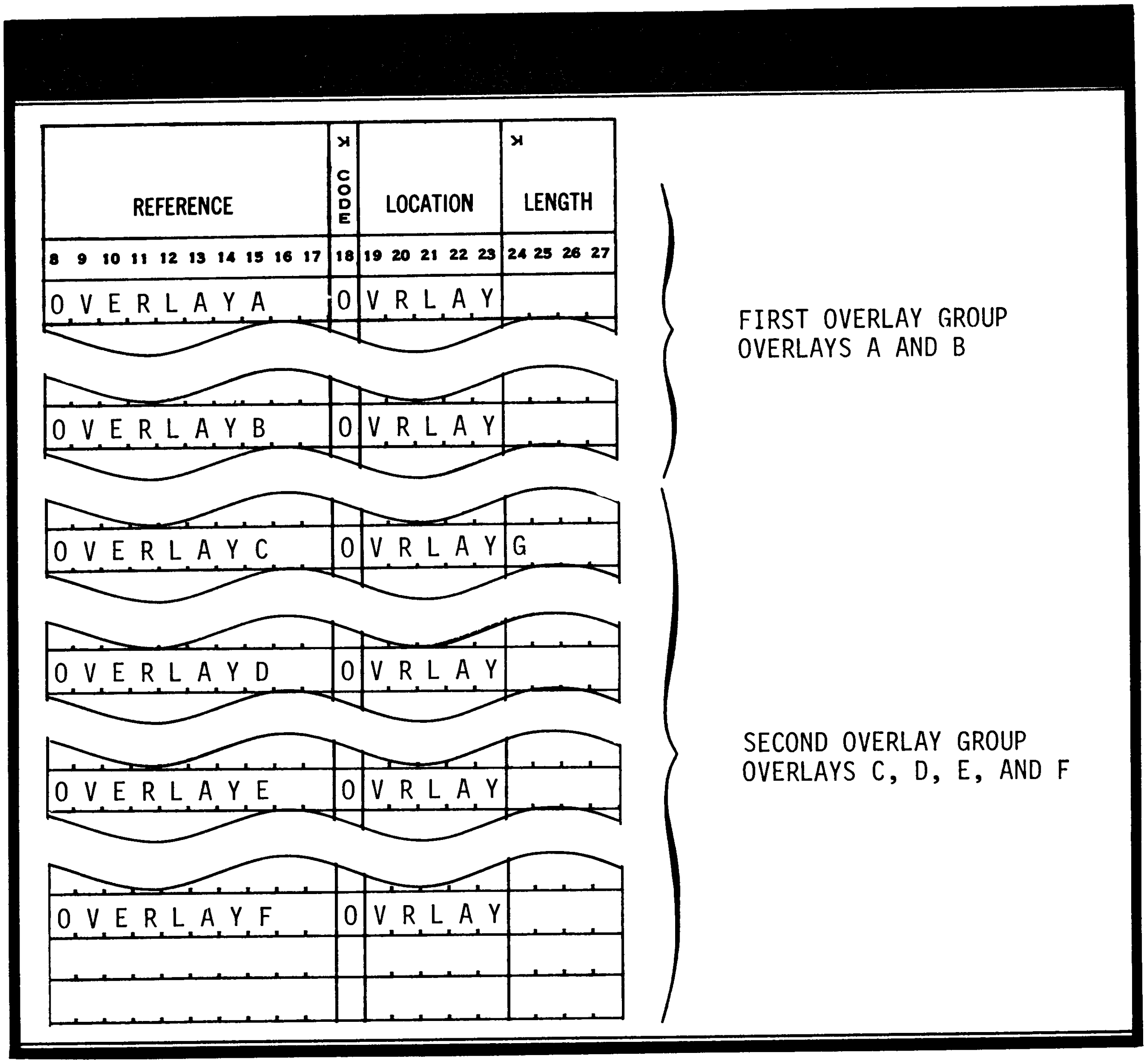
A group size if required, is entered as the second operand beginning in position 26.

In both of the preceding illustrations, the OVRLAYG instruction is used to begin a new overlay group, and specifies a qualifier OVERTIME for all references within the first overlay. During compilation, an area 512 bytes long is reserved on disc for the overlay group.

Conventions

All of the conventions listed for the OVRLAY instruction are applicable to the OVRLAYG instruction.

As illustrated below, the first OVRLAY instruction in a source program always starts a group of overlays, and only subsequent overlay groups need to be started with an OVRLAYG instruction.



| REFERENCE 8–17 | K CODE 18 | LOCATION 19–23 | K LENGTH 24–27 |
|---|---|---|---|
| OVERLAYA | O | VRLAY | |
| OVERLAYB | O | VRLAY | |
| OVERLAYC | O | VRLAY | G |
| OVERLAYD | O | VRLAY | |
| OVERLAYE | O | VRLAY | |
| OVERLAYF | O | VRLAY | |

If partial or independent overlay compilations are to be performed, the OVRLAYG instruction is used to start each overlay, and the overlay group size must be specified. Partial and independent overlay compilations are discussed in a separate publication, (see NEAT/3 REFERENCE MANUAL, COMPILATION PROCESS, tab 2, "Independent Overlay Compilation").

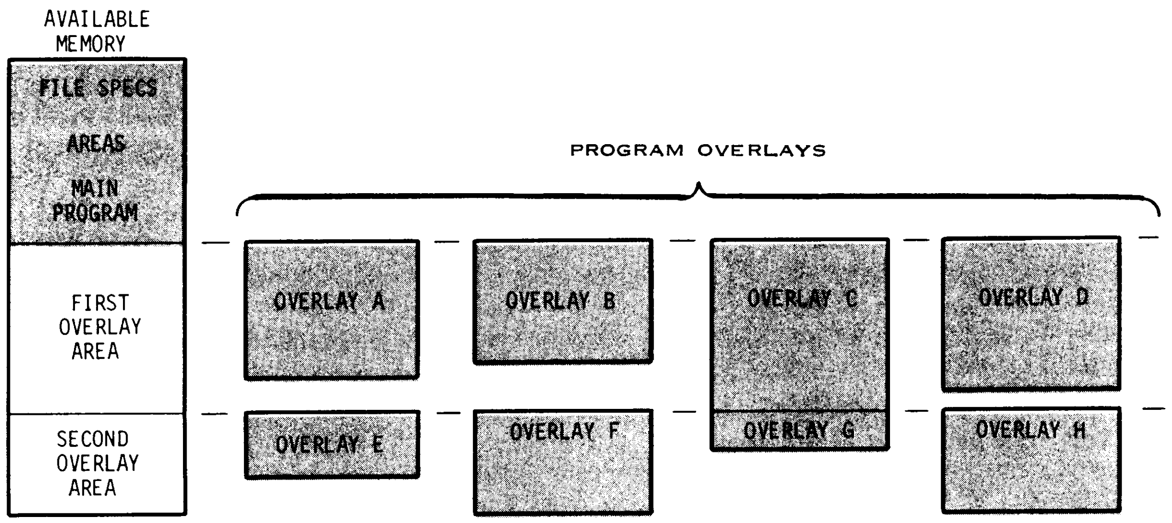
## PROGRAMMING CONSIDERATIONS FOR OVERLAYS

The use of dual disc units in the NCR Century Series enable the programmer to make efficient use of program overlays. The overlays are available almost instantly throughout the running of a program. However, to achieve the shortest program execution time, the program should call for new overlays as infrequently as possible.

The programmer can usually minimize the calling of new overlays by organizing his program so that branching is confined for as long as possible to the main program and those overlays already in memory.

The following simplified example illustrates how the various functions of a payroll program could be organized into overlays.

EXAMPLE OF PROGRAM OVERLAYS IN A PAYROLL PROGRAM



FUNCTIONS OF MAIN PROGRAM AND OVERLAYS

MAIN PROGRAM - CALCULATE PAY

| | |
|---|---|
| GROUP 1 | OVERLAY A - CALCULATE PAY |
| | OVERLAY B - CALCULATE TAX |
| | OVERLAY C - CALCULATE DEDUCTIONS |
| | OVERLAY D - PRINT CHECK |
| GROUP 2 | OVERLAY E - CALCULATE HOURLY RATE |
| | OVERLAY F - CALCULATE HOURLY RATE + COMMISSION |
| | OVERLAY G - CALCULATE SALARY |
| | OVERLAY H - CALCULATE SALARY + COMMISSION |

The records in the master and transaction files for this payroll run are organized in the following sequence:

- hourly-rated employees
- hourly-rated-plus-commission employees
- salaried employees
- salaried-plus-commission employees

The processing of every pay check requires overlays A, B, C, and D (group 1) in sequence. Overlay A always uses one of the four overlays in group 2. Because of the above record organization, overlay E is called into memory first and remains there until the pay checks for all the hourly-rated employees have been processed. Then, overlays F, G, and H are called to process the payroll checks of the other three categories of employees. During the entire payroll run, the four overlays in group 2 need only be called once each.

## ENTRY CONTROL INSTRUCTION

### ENTRY

### Function

The ENTRY control instruction has two functions, depending on whether the instruction is part of a main program or part of a program overlay.

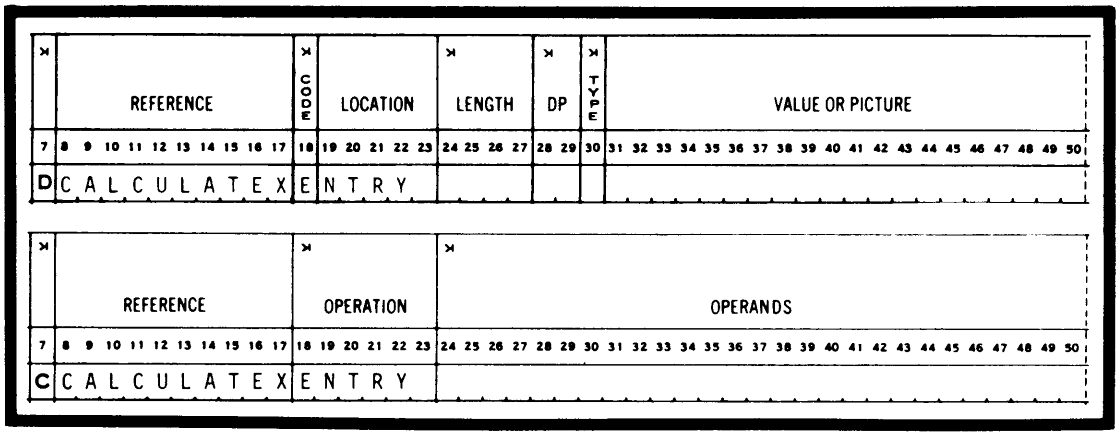
- An ENTRY instruction in the main program specifies a starting point for the program.
- An ENTRY instruction in a program overlay establishes an entry point to which the main program or other overlays may branch.

A main program may contain several sequential ENTRY instructions to specify several possible starting points; a program overlay may contain several sequential ENTRY instructions to specify several possible entry points into the overlay.

Each ENTRY instruction names in its REFERENCE column a reference tag which must be duplicated in the REFERENCE column of one of the subsequent instructions within the same logical division of the program (main program or overlay).

### Example

The ENTRY instruction may be coded on either a data layout sheet or a coding sheet, depending on where it is used. The use of each type of statement is discussed under Conventions in this publication.



| | REFERENCE | CODE | LOCATION | LENGTH | DP | TYPE | VALUE OR PICTURE |
|---|---|---|---|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 | 24 25 26 27 | 28 29 | 30 | 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| D | C A L C U L A T E X | E | N T R Y | | | | |

| | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | C A L C U L A T E X | E N T R Y | |

The ENTRY instruction must specify a reference as an entry point which corresponds to a subsequent instruction with the same reference. In the previous examples, the reference is CALCULATEX.

## Conventions

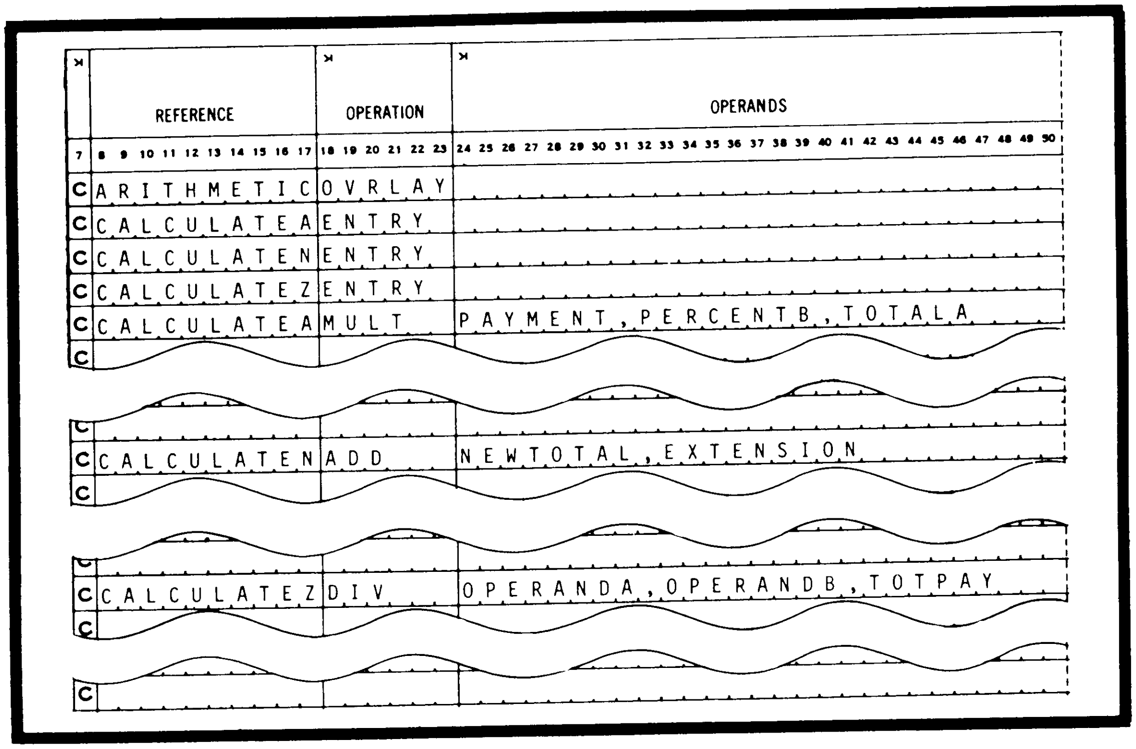
In a main program, the ENTRY instruction(s) must appear immediately following the compiler control statement and must be coded on a data layout sheet.

NOTE

> Since the main program always contains data definitions and ENTRY instructions precede data, the ENTRY instruction must be a data statement. The NEAT/3 Compiler does not accept coding statements before data statements.

In a program overlay, the ENTRY instruction must appear immediately following the OVRLAY (or OVRLAYG) instruction. If the overlay contains its own data definitions, the OVRLAY and ENTRY instructions must be coded on data layout sheets. If the overlay does not contain its own data definitions, the ENTRY instructions may be coded on coding sheets. The ENTRY instruction is always the same type of statement as the OVRLAY instruction that it follows.

The following sample coding establishes three entry points into a program overlay.



|  | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | ARITHMETIC | OVRLAY |  |
| C | CALCULATEA | ENTRY |  |
| C | CALCULATEN | ENTRY |  |
| C | CALCULATEZ | ENTRY |  |
| C | CALCULATEA | MULT | PAYMENT,PERCENTB,TOTALA |
| C |  |  |  |
| C |  |  |  |
| C | CALCULATEN | ADD | NEWTOTAL,EXTENSION |
| C |  |  |  |
| C |  |  |  |
| C | CALCULATEZ | DIV | OPERANDA,OPERANDB,TOTPAY |
| C |  |  |  |
| C |  |  |  |

In the preceding example, if CALCULATEN is a unique entry point in the entire program, the programmer simply branches to CALCULATEN. However, if the programmer has no assurance that CALCULATEN is not listed as another entry point somewhere in the program, he should specify the name of the overlay and the entry point reference (ARITHMETIC, CALCULATEN) in his branch instruction.

Only one instruction within each overlay may have the same reference as an ENTRY control instruction.

An ENTRY instruction may not be preceded by a renaming instruction (designated by an asterisk in position 18).

## SECTION CONTROL INSTRUCTION

SECT

Function

The SECT control instruction divides a program into sections. Because the NEAT/3 Compiler considers each section individually, several programmers may work on individual program sections without concern for duplication of reference tags between sections. All reference tags must be unique within each section.

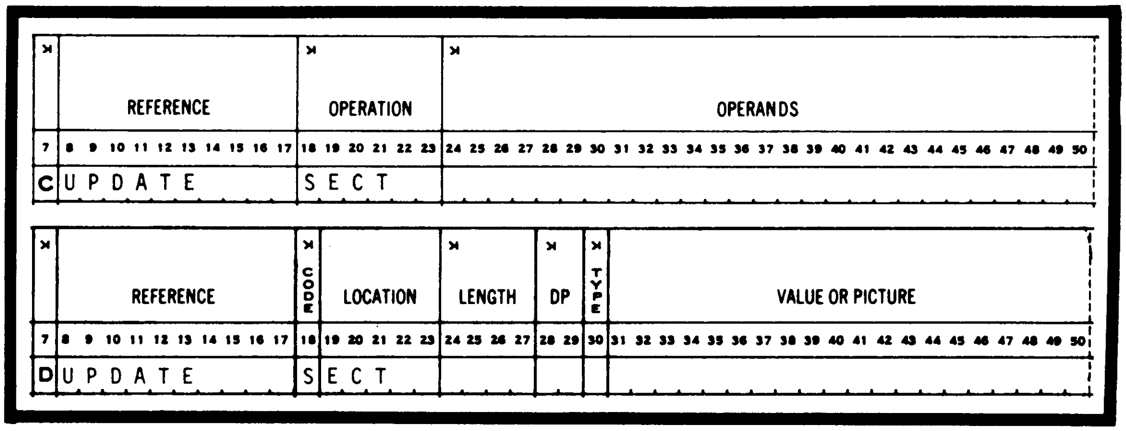
The SECT control instruction in a source program indicates to the NEAT/3 Compiler that a new program section is to be started.

NOTE

> The overlay instructions perform the same functions as the SECT instruction, in addition to their specific functions. The SECT instruction may be used in an overlay to divide the overlay into sections.

Example

The SECT instruction may be coded on either a data layout sheet or a coding sheet.



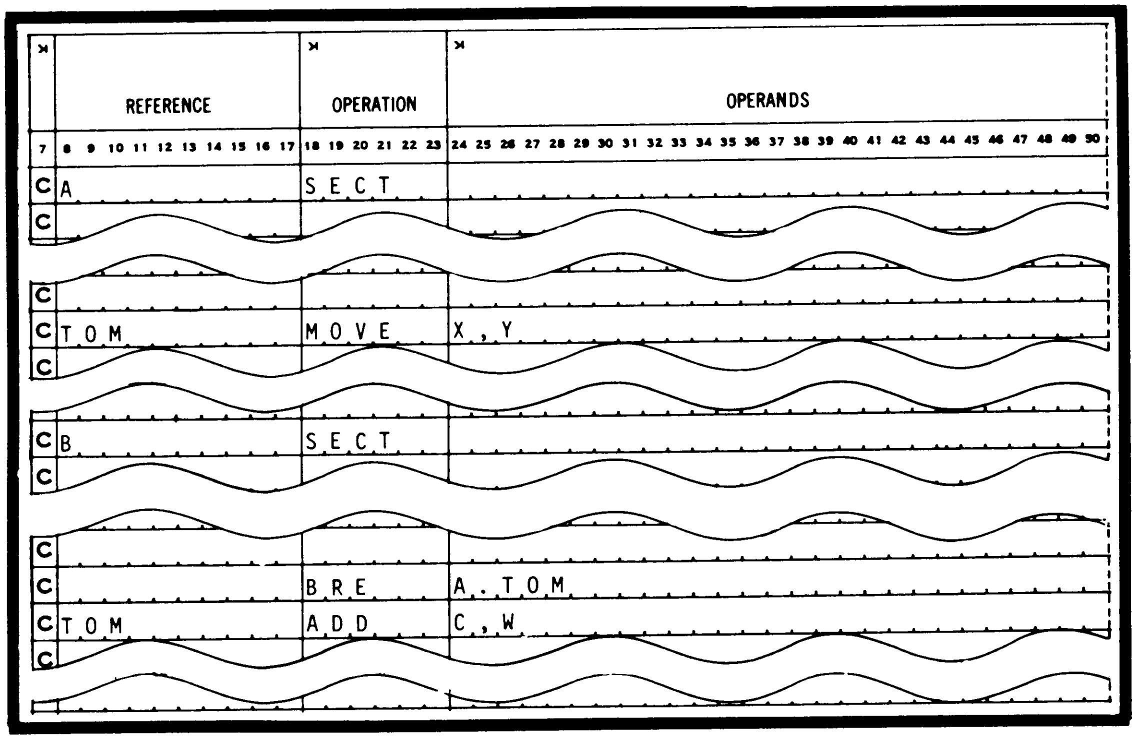
| ✕ | ✕ REFERENCE | ✕ OPERATION | ✕ OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | U P D A T E | S E C T | |

| ✕ | ✕ REFERENCE | ✕ CODE | LOCATION | ✕ LENGTH | ✕ DP | ✕ TYPE | VALUE OR PICTURE |
|---|---|---|---|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 | 24 25 26 27 | 28 29 | 30 | 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| D | U P D A T E | S | E C T | | | | |

The above SECT control instruction starts a new program section. The entry UPDATE in the REFERENCE column assigns the qualifier UPDATE to all references in this new program section.

Conventions

If the programmer references another program section in the operands column, he must qualify the reference with the name of the other section unless the reference is unique to the entire program.



| | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | A | SECT | |
| C | | | |
| C | | | |
| C | TOM | MOVE | X,Y |
| C | | | |
| C | B | SECT | |
| C | | | |
| C | | | |
| C | | BRE | A.TOM |
| C | TOM | ADD | C,W |
| C | | | |

In the above example, the same reference (TOM) is used in two different sections (A and B) of the program. TOM qualified by A is not the same reference as TOM qualified by B.

## RENAME CONTROL INSTRUCTION

### * (RENAME)

An asterisk (*) in column 18 of a coding sheet or a data layout sheet indicates a rename control instruction.

### Function

The programmer may use rename control instructions to assign a reference tag (or reference tags) to any desired point in a source program.

If a rename instruction precedes a source statement that does not have its own reference, the reference of the rename instruction becomes the reference of the source instruction. If a rename instruction precedes a source statement that has its own reference, the source statement in effect has two references either of which may be used as the operand of other instructions in the program.

The programmer may also precede a source statement with several rename instructions. In this case, all the references of the preceding rename instructions become associated with the source statement and may be used as the operand of other instructions. (See the following example).

In contrast to regular reference tags assigned to source statements, the reference tags assigned by the rename instruction do not start new program regions and do not establish barriers between local tags.

### Example

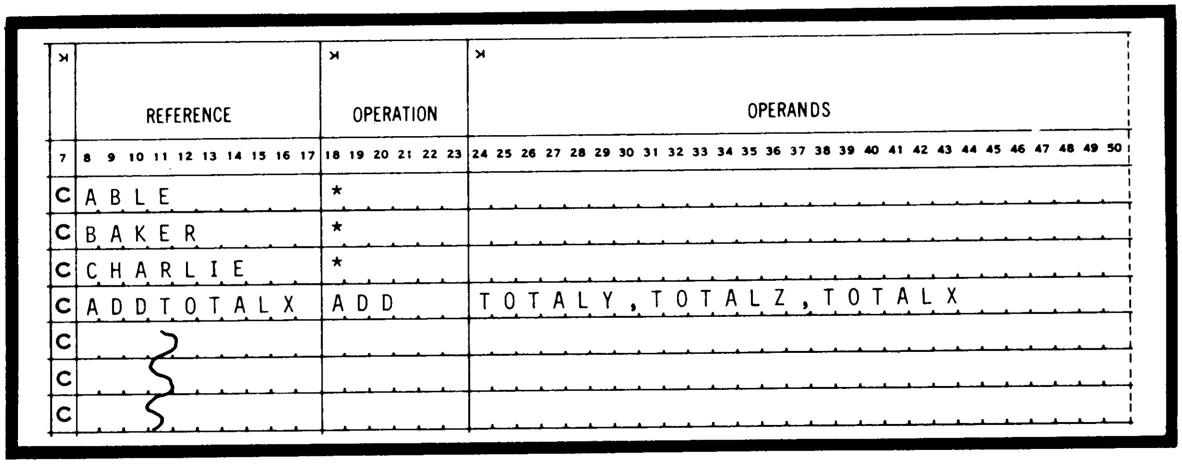
|  | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | REDUCEQ | * |  |

The above rename control instruction assigns the reference REDUCEQ to the source statement immediately following the control instruction.

## Conventions

The rename control instruction must immediately precede the source statement that is to assume the reference name of the rename instruction. If several different references are to be assigned to a source statement, several rename instructions may appear in succession immediately preceding that source statement.

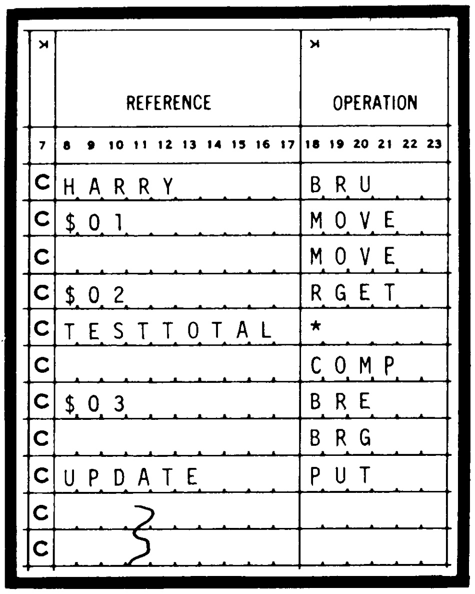
The following sample coding assigns three additional references to the ADD instruction that already has its own reference.



| | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| C | ABLE | * | |
| C | BAKER | * | |
| C | CHARLIE | * | |
| C | ADDTOTALX | ADD | TOTALY,TOTALZ,TOTALX |
| C | | | |
| C | | | |
| C | | | |

To branch to the ADD instruction in the preceding example, the programmer may name any one of the following references in the OPERANDS column of a branch instruction: ABLE, BAKER, CHARLIE, or ADDTOTALX.

The rename instruction is also frequently used in altering a recompilation master. The following sample coding illustrates how the use of the rename instruction permits the addition of a reference to a source statement without disrupting the affected program region and the local tags within that region.



| | REFERENCE | OPERATION |
|---|---|---|
| 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 |
| C | HARRY | BRU |
| C | $01 | MOVE |
| C | | MOVE |
| C | $02 | RGET |
| C | TESTTOTAL | * |
| C | | COMP |
| C | $03 | BRE |
| C | | BRG |
| C | UPDATE | PUT |
| C | | |
| C | | |

After recompilation, the rename (*) instruction and a COMP instruction appear in the source program listing. The COMP instruction has the effective reference TESTTOTAL. Branching to local tags $01, $02, and $03 within the formerly established program region is not affected.

A rename instruction may not precede an ENTRY instruction.

A local tag may not be used as the reference of a rename instruction.

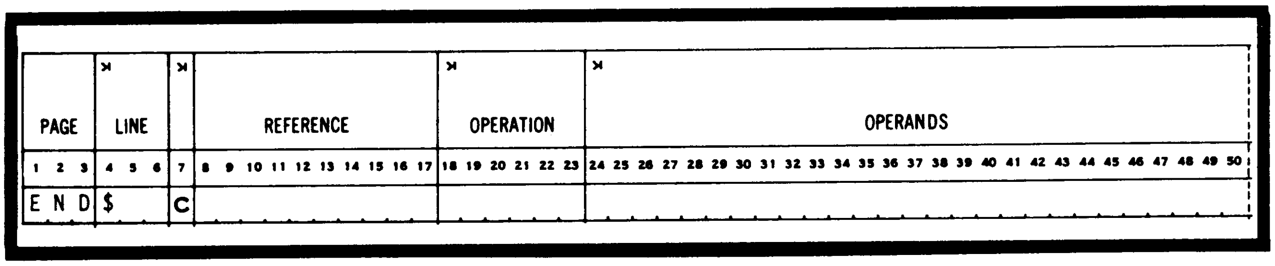
## END CONTROL INSTRUCTION

### END$

#### Function

The END$ control instruction indicates the end of a source program and notifies SPUR that no more source statements are to be read.

#### Example



| PAGE | LINE | | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|---|---|
| 1 2 3 | 4 5 6 | 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| E N D | $ | C | | | |

#### Conventions

The END$ control instruction must be entered in columns 1 through 4 of a coding sheet in place of the page and line number.

The END$ instruction must be the last source statement of each source program input to SPUR.

## SETPL CONTROL INSTRUCTION

SETPL

### Function

The SETPL control instruction specifies a new page and line number for the next source statement to be output to the disc file. This instruction is permitted at any time during a SPUR renumbering function.

By using the SETPL instruction, the programmer can separate sections of the program for convenient referencing of the compiler listing. To obtain this separation, the programmer must anticipate the increase effected by the SPUR renumbering function and assign the new page/line number accordingly.

The operands field of the SETPL instruction contains the page and line number to be assigned. SPUR executes only those SETPL instructions which specify a new page/line number greater than the new page/line number of the last statement output. If the instruction is executed, the SETPL statement and the comment NEW PAGE/LINE NUMBER appear on the compiler listing. If the instruction is not executed, the SETPL statement and the comment NO NEW PAGE/LINE NUMBER appear on the compiler listing.

### Example

| PAGE | LINE | | REFERENCE | OPERATION | OPERANDS |
|---|---|---|---|---|---|
| 1 2 3 | 4 5 6 | 7 | 8 9 10 11 12 13 14 15 16 17 | 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| 032 | 190 | C | | SETPL | 040000 |

The above SETPL instruction specifies that 040000 is to replace the new page/line number of the next source statement output to the disc file.

### Conventions

- The SETPL instruction may be used only if renumbering of source statements was requested on the compiler specification worksheet. Use of the SETPL instruction is permitted at any time during the SPUR renumbering function.
- A SETPL instruction will not be executed if the page/line number specified is less than or equal to the assigned page/line number of the last source statement output to the new recompilation master file.

- The SETPL instruction itself is not output to the disc file.
- Only one alphabetic character, in the leftmost position, is permitted in the page/line number specified by the SETPL instruction's operand. A SETPL instruction with an alphabetic character in the leftmost position of its operand results in an assigned page/line number consisting of that alphabetic character followed by five zeros.

# TABLE CONCEPTS

## INTRODUCTION

Generally, any information organized into a systematic arrangement (usually in rows or columns) that permits the quick lookup and reference of data can be described as a table. Because tables are perhaps the most convenient and efficient means of data organization, their use can frequently simplify the programming task.

The NEAT/3 language offers a variety of table structures, applications, and processing techniques. The purpose of this publication is to present a comparative overview of these structures and applications through the basic terms and concepts of table organization from which the user may determine the optimal table structure and processing technique for his individual requirements.

The second publication in this section considers the compiler worksheets necessary to describe and define a table and the organization of source-line information.

The following publications in this section are organized into two series. The first series contains one publication for each defined table structure available in the NEAT/3 language. Each describes the table structure, its processing technique and applications, and the conventions concerning its use. Since table structure and processing techniques determine how a table is constructed and how items are deleted from or inserted into the table, examples are included as appropriate for the TSERT and/or TBILDD and TDEL instructions.

The second series of publications in this section of the manual document in full detail the execution of each table instruction. Examples of how these instructions work are included in their respective publications.

## DEFINITIONS

### Table

A table is a series of identical, fixed-length items organized into a systematic arrangement to provide the efficient storage and reference of data. The organization of items within a table is similar to that of records within a file.

The following excerpt of a table organizes all the merchandise of a department store by department code and stock number.

| Department Code | Stock Number | [illegible] | [illegible] |
|---|---|---|---|
| 19 | 036924 | 0003 | 0027 |
| 19 | 050043 | 0044 | 0006 |
| 21 | 009110 | 0111 | 0014 |

## Item

In the table above, each line of data is considered an item. Information contained in a table must be organized as identical, fixed-length items. Table items, like file records, consist of one or more fields of related data or information. For example, in department 21, the inventory for stock number 009110 shows 14 units sold and 111 units in stock. To help identify and access the items of a table, an item field may be designated as a key.

## Key

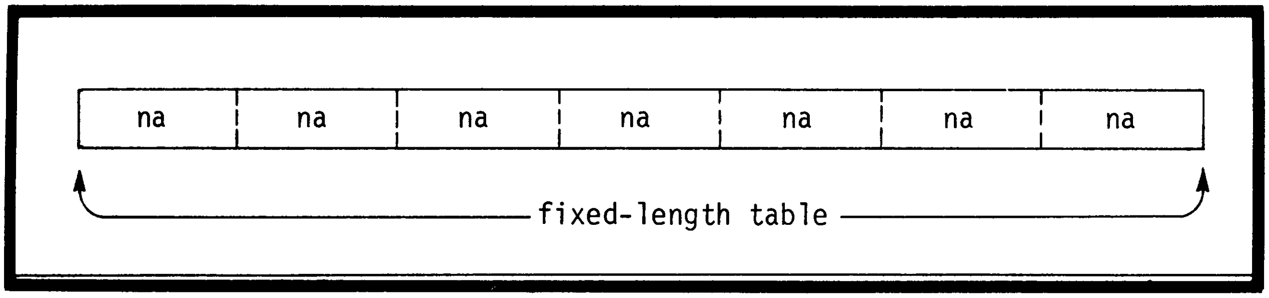
NEAT/3 tables permit the use of two key designations. The first or major key identifies and locates all the items of a particular group of related items. The second or minor key identifies and locates a specific item from the group of items identified by the major key. In the table above, department code as the major key identifies and locates all items for a particular department, e.g. department 19. Stock number as the minor key further identifies and locates the stock desired, e.g. stock number 036924. The designation of a minor key is optional.

## Length

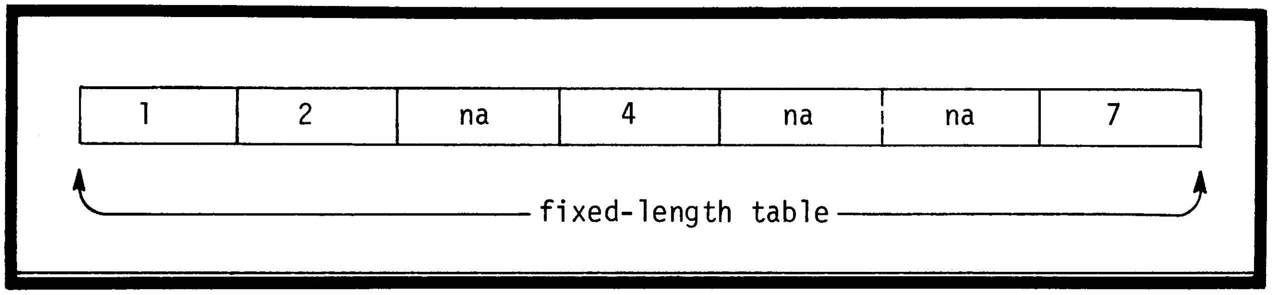
The overall length of a table is described as either fixed-length or variable-length and refers to the number of items stored in the table.

- Fixed-length Tables

  The overall length of a fixed-length table and the number of items in the table does not vary during processing. For example, assume at the time a fixed-length table is built, it contains no stored information. In effect, the table exists and is filled with identical, fixed-length items which are all designated as nonactive items.

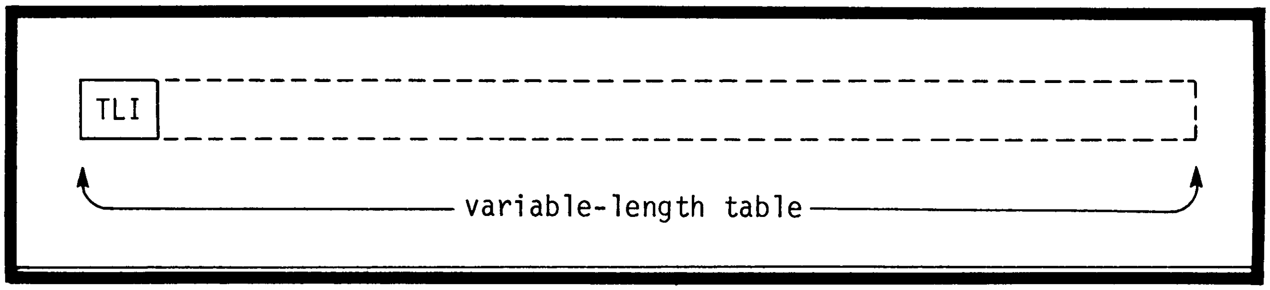


During processing, data may be entered into any nonactive item location, allowing nonactive items to exist between stored data, or active items. A fixed-length table may be built in an area, a record, or an item.

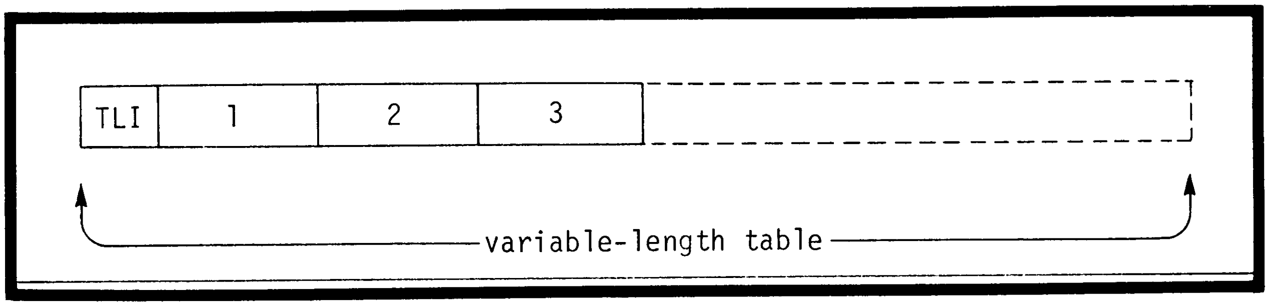


- Variable-length Tables

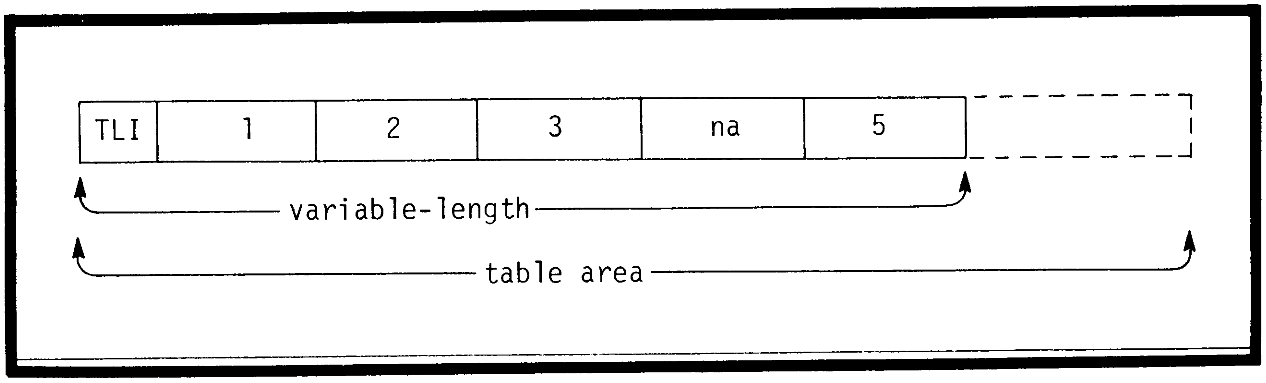
  The overall length of a variable-length table and the number of items in the table may be increased or decreased during processing. The first field (two 8-bit characters) of a variable-length table must be defined as the table length indicator (TLI). The TLI contains a binary number that specifies the total number of characters (including the 2 characters of the TLI) currently in the table. For example, assume at the time a variable-length table is built, it contains no stored information. In effect, only the first field containing the TLI (indicating a length of 2) exists.



During processing, as data items are placed into a variable-length table, they always occupy the beginning of the table and remain contiguous from the first item location.



This suggests that all items in a variable-length table are active items. However, an exception exists that allows for nonactive items to occur in a variable-length table in an area. For example, when an item is to be inserted at an item location where no active item precedes it, the table expands to accommodate for the nonactive item location before inserting the new active item.



A variable-length table may be built in either an area or a record.

## Table Handling

There are two table-handling techniques: slot processing and pushdown processing.

- Slot Processing

  Slot processing is a table-handling technique in which software allows items to be added to or deleted from the table by considering each item to be either active or nonactive. All fixed-length tables use slot processing. In addition, slot processing may be applied to the handling of a variable-length table in an area as described above where the table expands to accommodate a nonactive location before inserting a new data item. Examples of slot insertion and deletion are included in the separate publications for each of the defined table structures.

- Pushdown Processing

  Pushdown processing is a table handling technique in which the overall length of the table and the number of items in the table increases or decreases as items are added to or deleted from the table. Pushdown processing can only be used with variable-length tables. Examples of pushdown insertion and deletion are included in the separate publications for each of the defined table structures.

## FUNCTIONS

Table functions are handled by specific table instructions which may be separated into the five general categories that follow:

- To Initialize a Table

  Before any table processing can begin, the programmer must initialize the table. The variations of the TBEG instruction initialize the table locations and counters so that either the entire table may be built (TBEGB - begin to build) or the items in the existing table may be accessed (TBEGF - begin to find).

- To Build a Table

  If the table is to be built, the programmer codes the applicable variation of the TBILD instruction which constructs the table in either a direct (TBILDD - build direct) or sequential (TBILDN - build next) manner.

- To Access Items Within the Table

  If a particular item in the table is to be accessed, the programmer codes the applicable variation of the TFIND instruction which finds the specific item by a random search (TFINDR - find random), a sequential search by key comparison (TFINDS - find sequential, TFINDP - find previous sequential), a binary search (TFINDB - find binary), a sequential search by location (TFINDN - find next, TFINDO - find next in order after last table instruction), or by the direct location of a specific item (TFINDD - find direct).

- To Delete Items From and To Add Items To a Table

  When an item is to be deleted from or inserted into a table, the programmer codes either a TDEL (delete), a TSERT (insert), or a TBILDD (build direct) instruction.

- To Perform Special Functions Upon a Table

  At the programmer's option, various other table instructions may be coded which perform unique but essential tasks in table maintenance. These instructions are TSHIFT, TPACK, TSORT, TMARK, TRESET, and TJUMP.

The programmer decides which functions his program requires and codes the applicable table instructions. Examples using these table instructions are found in their separate publications.

In some instances, table instructions are restricted by table structure. The following illustration shows which instructions are applicable to each of the 6 table structures.

| STRUCTURE 1 | STRUCTURE 2 | STRUCTURE 3 | STRUCTURE 4 | STRUCTURE 5 | STRUCTURE 6 |
|---|---|---|---|---|---|
| TBEGB | TBEGB |  | TBEGB | TBEGB | TBEGB |
| TBEGF | TBEGF | TBEGF | TBEGF | TBEGF | TBEGF |
| TBILDN | TBILDN |  | TBILDN | TBILDN | TBILDN |
| TBILDD | TBILDD |  | TBILDD |  |  |
| TFINDN | TFINDN | TFINDN | TFINDN | TFINDN | TFINDN |
| TFINDD | TFINDD | TFINDD | TFINDD |  |  |
| TFINDR | TFINDR | TFINDR | TFINDR | TFINDR | TFINDR |
| TFINDS | TFINDS | TFINDS | TFINDS | TFINDS | TFINDS |
| TFINDB | TFINDB | TFINDB | TFINDB | TFINDB | TFINDB |
| TFINDP | TFINDP | TFINDP | TFINDP | TFINDP | TFINDP |
| TFINDO | TFINDO | TFINDO | TFINDO | TFINDO | TFINDO |
| TDEL | TDEL | TDEL | TDEL | TDEL | TDEL |
|  |  |  | TSERT | TSERT | TSERT |
| TPACK | TPACK | TPACK | TPACK |  |  |
| TSORTA | TSORTA | TSORTA | TSORTA | TSORTA | TSORTA |
| TSORTD | TSORTD | TSORTD | TSORTD | TSORTD | TSORTD |
| TSHIFT | TSHIFT | TSHIFT | TSHIFT | TSHIFT | TSHIFT |
| TMARK | TMARK | TMARK | TMARK | TMARK | TMARK |
| TRESET | TRESET | TRESET | TRESET | TRESET | TRESET |
| TJUMP | TJUMP | TJUMP | TJUMP | TJUMP | TJUMP |

## TABLE STRUCTURES

Table location, length, and handling techniques make available the six defined structures that follow:

- Structure 1 - A fixed-length table in an area, using slot processing.
- Structure 2 - A fixed-length table in a record, using slot processing.
- Structure 3 - A fixed-length table in an item of another table, using slot processing.
- Structure 4 - A variable-length table in an area, using slot processing.
- Structure 5 - A variable-length table in an area, using pushdown processing.
- Structure 6 - A variable-length table in a record, using pushdown processing.

## Freestanding Tables

Freestanding tables contain information common to many records or even to many files in a program. Three structures are possible for freestanding tables in an area of memory.

- Fixed-Length Table Within an Area Using Slot Processing (Structure 1)

  1. This structure works well for a table that has a high insertion/deletion rate of items.
  2. This structure can be built in either a direct or sequential manner.
  3. Items in this table structure are normally processed by their location within the table rather than by the contents of their keys.
  4. Since the TSERT instruction cannot be used in this structure, items are added or inserted using the TBILDD instruction.

- Variable-Length Table Within an Area Using Slot Processing (Structure 4)

  1. This structure works well for a table that has a high insertion/deletion rate of items.
  2. This structure can be built in either a direct or sequential manner.
  3. This table structure is suggested for applications in which items are processed by the contents of their keys.
  4. All table instructions are usable in this structure.

- Variable-Length Table Within an Area Using Pushdown Processing (Structure 5)

  1. This structure can only be built in a sequential manner, because the TBILDD instruction is not used with this structure.
  2. Items in this table structure are processed by the contents of their keys.
  3. Items in this structure cannot be accessed in a direct search.
  4. The TPACK instruction (which decreases access time of items) cannot be used in this structure.

## Record Tables

When each record in a file contains related information that can be organized into identical, fixed-length items within a field of each record, this information can be defined as a record table. One table definition (like one record definition) is all that is needed to access the many tables (in each record) contained in the file. Two structures are possible for record tables in a file. Conventions associated with each of these structures are outlined below.

- Fixed-Length Table Within a Record Using Slot Processing (Structure 2)

  1. This structure permits the simple step-through processing of a number of items in that field of a record defined as a table.
  2. This structure can be built in either a direct or sequential manner.
  3. Since the TSERT instruction cannot be used in this structure, items are added or inserted using either the TBILDD, TFINDN, COMP, or MOVE instructions.

- <u>Variable-Length Table Within a Record Using Pushdown Processing</u> (Structure 6)

  1. This structure permits the simple step-through processing of a number of items in a variable-length portion (field) of a record defined as a table and automatically updates the variable length indicator (VLI) of the record.
  2. The overall length of the record may increase or decrease as items are added to or deleted from the table portion of the record.
  3. This structure can only be built in a sequential manner, because the TBILDD instruction is not used with this structure.
  4. Items in this structure cannot be accessed in a direct search.
  5. The TPACK instruction (which decreases access time of items) cannot be used in this structure.

<u>Minor Tables</u>

If each item in one of the other five table structures contains many units of identically formatted data, these like units may be defined as items in a minor table. One minor table definition is all that is needed to subdivide the data in each item of the major table. A common structure is used for all minor tables regardless of the major table in which they exist. Conventions of this structure are outlined below.

- <u>Fixed-Length Table Within a Table Using Slot Processing</u> (Structure 3)

  1. Since the TBILD instructions cannot be used with this structure it must be constructed using data definitions.
  2. Since the TSERT instruction cannot be used in this structure, items are added or inserted using the TFINDN, COMP and MOVE instructions.

## TABLE WORKSHEETS

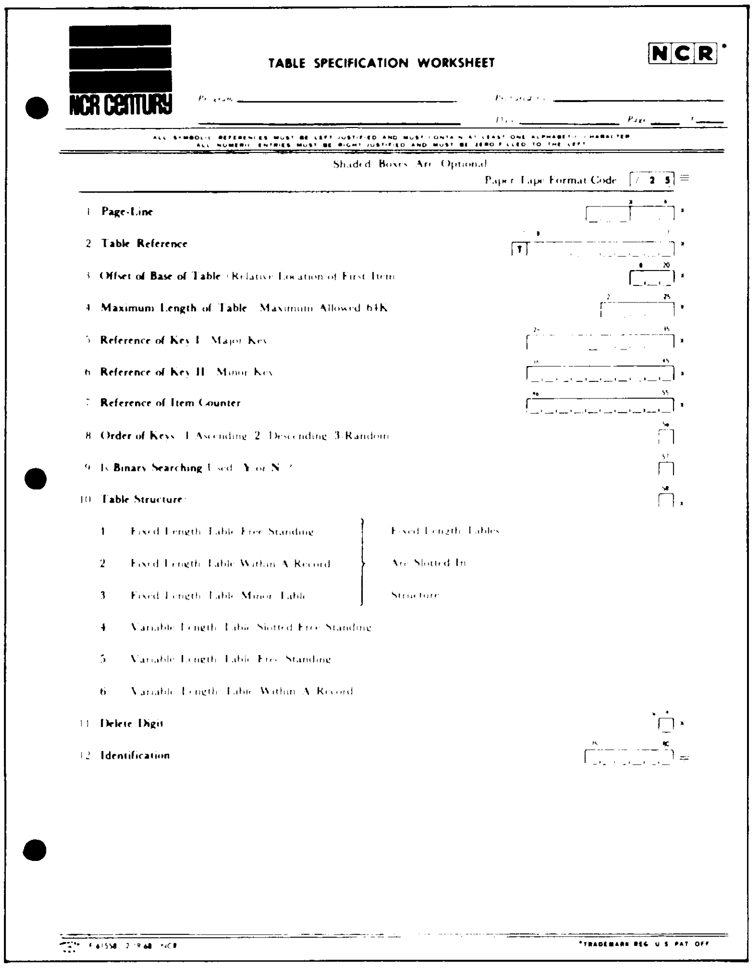
Two types of compiler worksheets are needed to describe and define a table:

1. A table specification worksheet is used to describe the characteristics of the table.
2. A data layout worksheet is used to define the length and format of the items and fields within the table.

Table instructions are organized on a coding worksheet.

### TABLE SPECIFICATION WORKSHEET

A table specification worksheet is illustrated below. The programmer uses one of these sheets to describe the characteristics of each table in his program.



NCR CENTURY

TABLE SPECIFICATION WORKSHEET

NCR*

Program ____________ Prepared by ____________

____________ Date ____________ Page ____ of ____

ALL SYMBOLIC REFERENCES MUST BE LEFT JUSTIFIED AND MUST CONTAIN AT LEAST ONE ALPHABETIC CHARACTER
ALL NUMERIC ENTRIES MUST BE RIGHT JUSTIFIED AND MUST BE ZERO FILLED TO THE LEFT

Shaded Boxes Are Optional

Paper Tape Format Code / 2 5

1 **Page-Line**

2 **Table Reference** T

3 **Offset of Base of Table** (Relative Location of First Item)

4 **Maximum Length of Table** (Maximum Allowed 64K)

5 **Reference of Key I** (Major Key)

6 **Reference of Key II** (Minor Key)

7 **Reference of Item Counter**

8 **Order of Keys** (1 Ascending 2 Descending 3 Random)

9 **Is Binary Searching** Used (Y or N)?

10 **Table Structure**:

| | | |
|---|---|---|
| 1 | Fixed Length Table Free Standing | Fixed Length Tables |
| 2 | Fixed Length Table Within A Record | Are Slotted In |
| 3 | Fixed Length Table Minor Table | Structure |
| 4 | Variable Length Table Slotted Free Standing | |
| 5 | Variable Length Table Free Standing | |
| 6 | Variable Length Table Within A Record | |

11 **Delete Digit**

12 **Identification**

F 61558 2/9/68 NCR

*TRADEMARK REG. U.S. PAT. OFF.

The programmer should fill in the header, page-line number (question 1), the delete digit (question 11), and the identification tag (question 12) as defined in NEAT/3 REFERENCE MANUAL, INTRODUCTION AND DATA, tab 3, "Programming Worksheets". The paper tape code must be punched if paper tape is used for input to the compiler.

1. **Page-Line** 1 6

2. **Table Reference** 7 8 17 T

The letter T in position 7 is preprinted on the table specification worksheet and must be punched.

Enter in positions 8 through 17 the name of the table. This name may contain from 1 to 10 characters which are made up of the letters A through Z and/or the numerals 0 through 9. The table name must begin in position 8 and must contain at least one alphabetical character. Spaces are not permitted within the table name.

3. **Offset of Base of Table** (Relative Location of First Item) 18 20

Enter in positions 18 through 20 the relative location of the first item in the table. This entry is optional for fixed-length tables (structures 1, 2, and 3). For variable length tables (structures 4, 5, and 6), remember to include in this count the two characters needed for the TLI.

4. **Maximum Length of Table** (Maximum Allowed 64K) 21 25

If this table is fixed-length, enter in positions 21 through 25 the number of characters this table contains.

If this table is variable-length, enter in positions 21 through 25 the maximum number of characters that this table is to contain. Remember to include the two characters for the TLI.

5. **Reference of Key I** (Major Key) 26 35

If this table has two keys, enter the reference name of the major key (the field whose contents identify a particular group of items in the table).

If this table has only one key, enter the reference name of the key (the field whose contents identify the desired item in the table).

If no key is to be used, tag the associated item and enter that reference name in this space.

**Reference of Key II** (Minor Key) 36 45

If this table has two keys, enter the reference name of the minor key (the field whose contents identify one item within the group narrowed by the major key).

If this table has one key, omit this entry.

**Reference of Item Counter** 46 55

If the optional item counter is desired, enter the reference of this counter.

If the optional item counter is not desired, leave these positions blank.

An item counter, an optional feature of the NEAT/3 tables, is a field that contains the number of items currently in the table. During program execution, the software increments and decrements this number as it adds and deletes items from the table.

If a running total of the number of items currently in the table is desired, define this counter on the data layout sheets. This counter must be an unsigned decimal field (type U) and must originally contain the initial number of items in the table.

- If the table is within a record, this field must also be in the record and must precede the table.
- If the table is within an area, this field may be anywhere in memory other than in the table.
- If the table is within another table, the location of the major table (either in a record or in an area) determines whether the counter field for the minor table is in a record or in an area.

**Order of Keys** (1-Ascending, 2- Descending, 3-Random) 56

Enter 1 if the contents of both the major and minor keys are in ascending sequence (low to high). The contents may be either numeric or alphanumeric.

Enter 2 if the contents of both the major and minor keys are in descending sequence (high to low). The contents may be either numeric or alphanumeric.

Enter 3 if the contents of both or one of the keys are in a random alphanumeric or random numeric sequence.

9. Is **Binary Searching** Used (**Y** or **N**)? 57 ☐

Enter Y if a binary search (TFINDB) is to be performed on this table at any time during the execution of this program.

Enter N if a binary search (TFINDB) is not to be performed on this table during the execution of this program.

10. **Table Structure:** 58 ☐

1 = Fixed Length Table-Free Standing

2 = Fixed Length Table-Within A Record

3 = Fixed Length Table-Minor Table

} Fixed Length Tables Are Slotted In Structure

4 = Variable Length Table Slotted-Free Standing

5 = Variable Length Table Free Standing

6 = Variable Length Table Within A Record

Enter the number from 1 to 6 which indicates the structure of the table.

11. **Delete Digit** 74 ☐

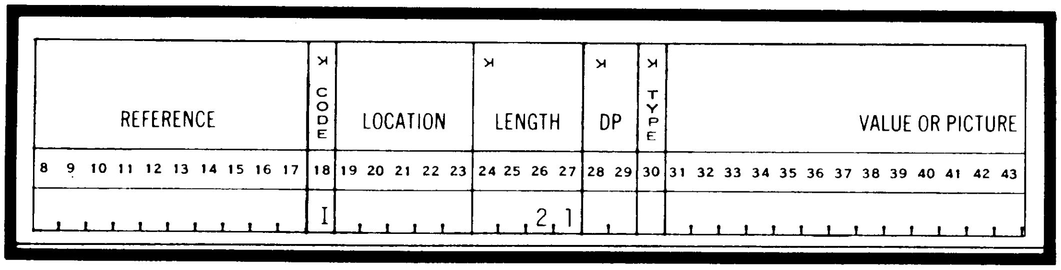
12. **Identification** 75–80 ☐☐☐☐☐☐

## DATA LAYOUT WORKSHEET

The data layout worksheet is used to define to the compiler the format of the items and fields within each table. Since the format of every item within each table must be identical, only one item and its fields need be defined for each table.

### Item Definition

When input to the compiler, the item definition must immediately follow its associated table-specification source line. Two entries are required for each item definition; in column 18, place an I and in columns 24-27, place the number of 8-bit characters in the item.

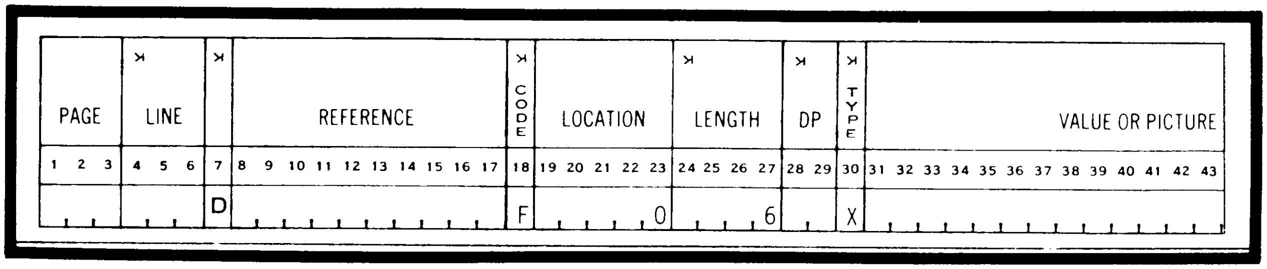


| REFERENCE | CODE | LOCATION | LENGTH | DP | TYPE | VALUE OR PICTURE |
|---|---|---|---|---|---|---|
| 8 9 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 | 24 25 26 27 | 28 29 | 30 | 31 32 33 34 35 36 37 38 39 40 41 42 43 |
| | I | | 21 | | | |

### Field Definitions

The field definitions are subdivisions of the item definition. They specify the units of data that make up the item. One (or two) of the fields is generally the key(s) referenced on the table specification worksheet, question 5 (and 6). The following entries are required for every field definition:

- In column 18, place an F.
- In columns 19-23, indicate the relative location of the field within the item.
- In columns 24-27, place the number of 8-bit characters in the field.
- In column 30, place the data type of the field.



| PAGE | LINE | | REFERENCE | CODE | LOCATION | LENGTH | DP | TYPE | VALUE OR PICTURE |
|---|---|---|---|---|---|---|---|---|---|
| 1 2 3 | 4 5 6 | 7 | 8 9 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 | 24 25 26 27 | 28 29 | 30 | 31 32 33 34 35 36 37 38 39 40 41 42 43 |
| | | D | | F | 0 | 6 | | X | |

Using the rules governing item and field definitions, a bank that tables each transaction to the depositor's master record defines the items and fields as follows:

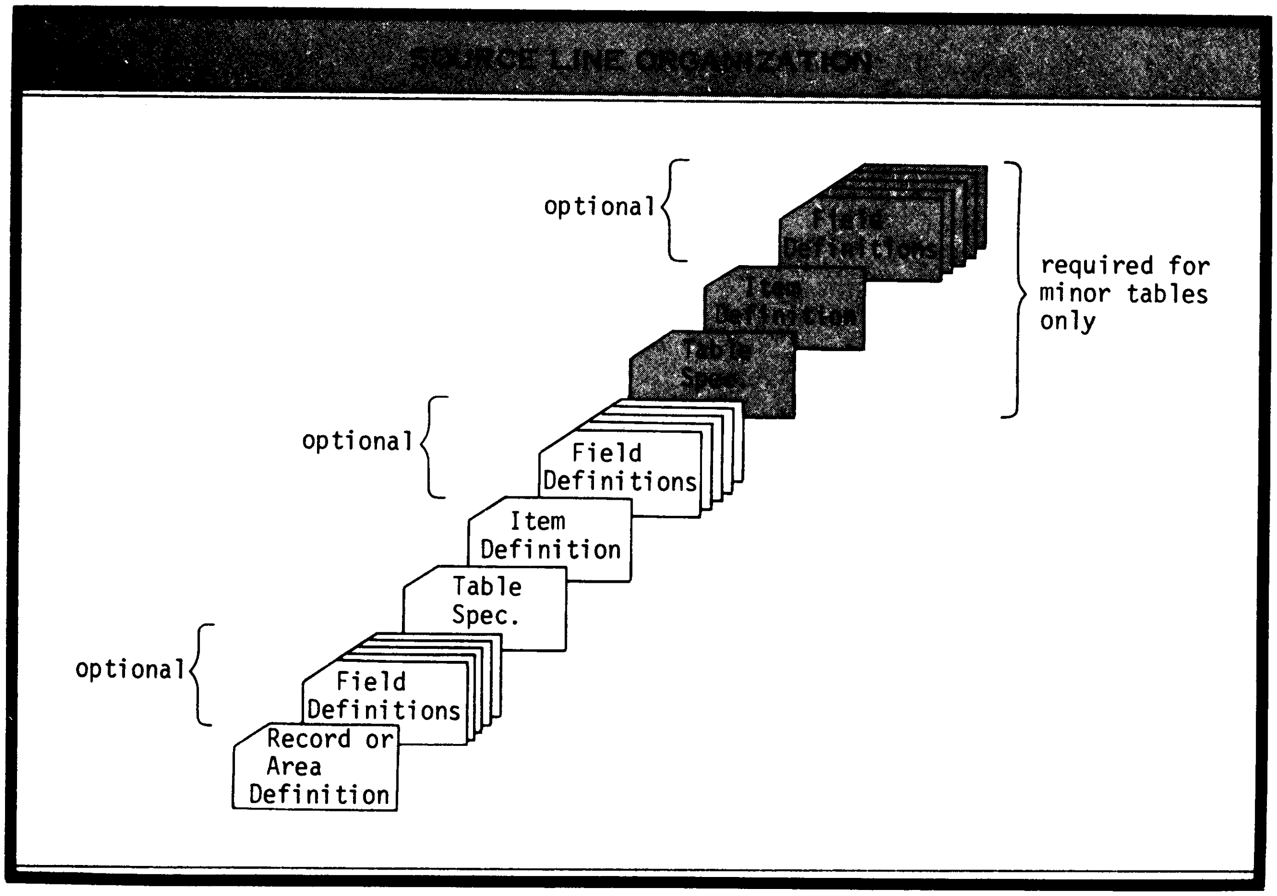
|  | X | X |  | X |  | X | X | X |  |
|---|---|---|---|---|---|---|---|---|---|
| PAGE | LINE |  | REFERENCE | CODE | LOCATION | LENGTH | DP | TYPE | VALUE OR PICTURE |
| 1 2 3 | 4 5 6 | 7 | 8 9 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 | 24 25 26 27 | 28 29 | 30 | 31 32 33 34 35 36 37 38 39 40 41 42 43 |
|  |  | D |  | I |  | 21 |  |  |  |
|  |  | D | DATE | F | 0 | 6 |  | X |  |
|  |  | D | CODE | F | 6 | 1 |  | X |  |
|  |  | D | AMOUNT | F | 7 | 7 | 2 | U |  |
|  |  | D | BALANCE | F | 14 | 7 | 2 | U |  |

The format of the above defined item would be as follows:

| Date |  |  | Transaction Code | Amount | Balance |
|---|---|---|---|---|---|
| Da | Mo | Yr |  |  |  |
| XX | XX | XX | X | XXXX XX | XXXX XX |

## SOURCE LINE ORGANIZATION

The following illustration shows the source-line sequence needed to properly define tables.



- A definition of the table environment (area or record) must be presented first.
- The table environment (area or record) may contain constants for working-storage fields. These field definitions are optional.
- The table specification sheet must be presented next.
- The item definition for the table must follow the table specification sheet.
- The field definitions within the item are presented following the item definition. These field definitions are optional.
- If a minor table (table within a table) exists, a table specification sheet for the minor table must be presented next.
- The item definition for the minor table must follow the minor table specification sheet.
- The field definitions within the item for the minor table follow the item definition. These field definitions are optional.

## FIXED—LENGTH TABLE WITHIN AN AREA

## USING THE SLOT PROCESSING TECHNIQUE

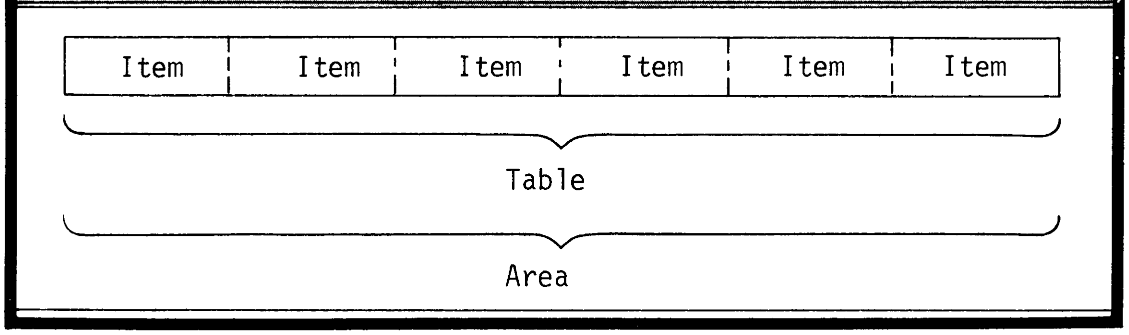
## ( STRUCTURE 1)

### DESCRIPTION

A fixed-length table is one whose overall length does not vary. During processing, a programmer may frequently wish to access a fixed-length table which contains information common to many records or even to many files. To do this, he may define this table to be freestanding in a memory area.

The following illustration shows the general format of a table within a memory area.

FORMAT OF A TABLE WITHIN A MEMORY AREA



The programmer's definition of this area must reserve enough memory to contain the entire table. The compiler assigns an area in memory to this table from the programmer's entries on the data layout sheets.

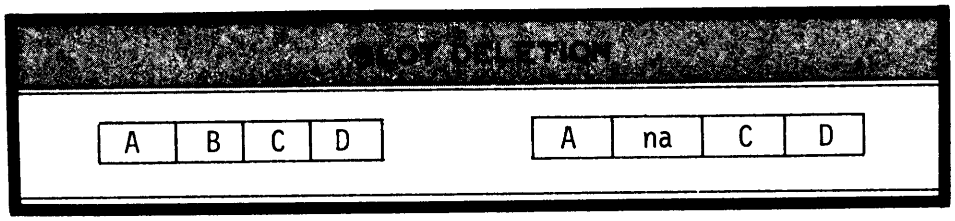
### SLOT PROCESSING TECHNIQUE

Slot processing is the only technique that can be used on fixed-length tables. Slot processing is a technique of table handling that allows items to be added to or deleted from the table not by altering the overall length of the table, but by simply considering each item to be either active or nonactive.

#### To Delete an Item Using TDEL

To delete an item from the table, the TDEL instruction places a code in the item. It now considers this item to be nonactive or deleted.

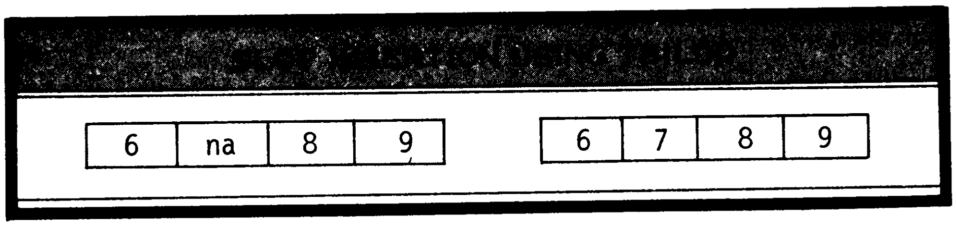
In the following example, TDEL deletes item B from the table using the slot processing technique. Note that the table size does not change.



To Insert an Item Using TBILDD

To insert an item into a table, TBILDD directly inserts the item by placing a new item into the location specified by the user. However, if the location is currently active, TBILDD does not insert the new item but sets the E flag and transfers control to the routine specified by the branch operand.

In the following example, TBILDD is to insert a new item into the second item position. This new item has a key of 7.

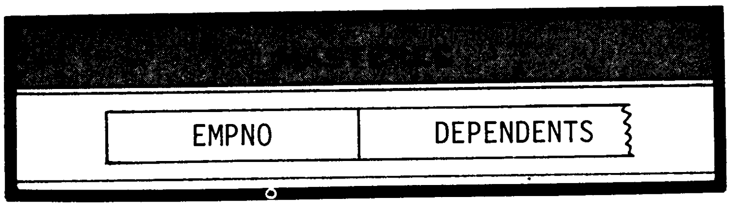


SAMPLE PROBLEM

A payroll program must access the master record for each employee, subtract the taxes and other deductions pertinent to the employee, and print a paycheck. The procedural instructions which perform these functions require the data definitions of the master record, the transaction record, and the tax table.

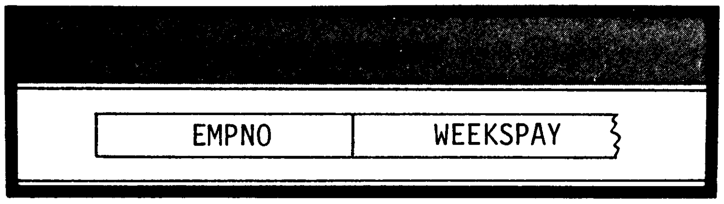
Master File

The amount of each employee's withholding tax is determined by his gross pay and the number of his dependents. The number of dependents for each employee is contained on a master record.



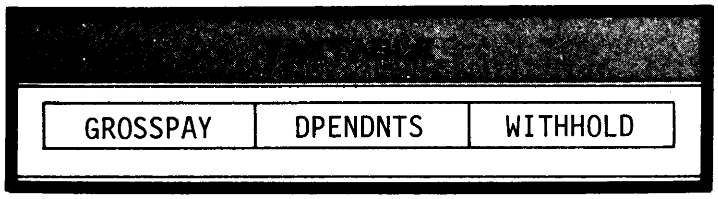
## Transaction File

Each employee's gross pay is contained on a transaction record.



## Tax Table

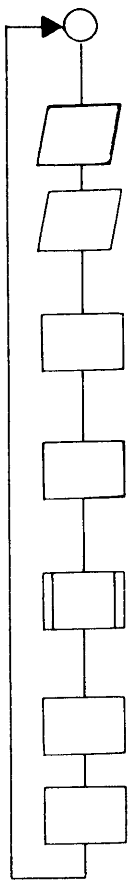
Since many employees fall within the same salary range and have the same number of dependents, the programmer defines a precalculated tax table as being freestanding in memory. This tax table can be accessed any time during processing. Each item in this freestanding tax table contains three fields: GROSSPAY, DPENDNTS, and WITHHOLD.



## Procedural Instructions

Using the contents of WEEKSPAY and DEPENDENTS, the program performs a table lookup for the item applicable to each employee. When this information is found, the program subtracts the withholding tax from the employee's salary.

After the program subtracts all other deductions pertinent to this master record, it prints the paycheck and branches to get the next transaction record. Consider the following flowchart.



| | |
|---|---|
| GET | A transaction record. |
| GET | The corresponding master record. |
| TFINDS | In the tax table the item whose major key equals the contents of WEEKSPAY and whose minor key equals the contents of DEPENDENTS. |
| SUB | The withholding tax from the contents of WEEKSPAY. |
| SUB | Other deductions pertinent to this master record. |
| PUT | Print the paycheck. |
| BR | To get the next transaction record. |

## BUILDING THE TABLE

A fixed-length table within an area (structure 1) may be initially built in any of three ways: through the execution of TBILDD, through the execution of TBILDN, or through data definitions. The method chosen to build the table depends upon table use.

The following guidelines are by no means definitive. They are merely suggested approaches to building a table. If a programmer has an application that will process well by combining some of the following suggestions, he is free to do so as long as he does not violate any conventions pertaining to table structure or instructional use, e.g. he may not in any circumstance use a TSERT instruction on a fixed-length table within an area.

## Building the Table with TBILDD

TBILDD is used to build a fixed-length table that is to be processed by item location and that has insertion/deletion capabilities.

Since all fixed-length tables within an area must use the slot processing technique, the only way items can be inserted into the table is through the execution of TBILDD. TBILDD does not look at the contents of the keys to determine where to build the item. It builds an item into the first, fourth, tenth, fiftieth, etc. item location in the table. This requires that the programmer know into which location the item is to be inserted.

Hence, if items are to be added to or deleted from the table, the programmer should process this table by item location rather than by key comparison. The following instructional variations logically complement each other and should be used when processing a table within an area that has insertion/deletion capabilities:

- To build the table, use TBILDD.
- To access items within the table, use TFINDD.
- To insert items into the table, use TBILDD.
- To delete items from the table, use TDEL.

## Building the Table with TBILDN

TBILDN is used to build during program execution a fixed-length table that is to be processed by key comparison but that has no insertion/deletion capabilities.

A programmer may wish to build a fixed-length table within an area during program execution. The keys of this table are constant for today's run, i.e. the program will never insert items into or delete items from the table once the table is built. Tomorrow, however, an entirely new table may be built whose keys may or may not resemble today's keys. (The data comprising the actual contents of the keys may originate from a COT reader or from magnetic media as an output of another program.)

The following instructional variations complement each other and should be used when processing a fixed-length table by key comparison:

- To build the table, use TBILDN.
- To access items within the table:
  - Use TFINDN or TFINDP if the items are to be accessed sequentially (one after the other).
  - Use TFINDR if the items are to be accessed randomly and if the keys are not sequentially organized.
  - Use TFINDB, TFINDS, or TFINDO if the items are to be accessed randomly and if the keys are sequentially organized.

## Building the Table with Data Definitions

Data definitions may be used to build into an object program a fixed-length table which is to reside in a memory area during program execution. Each time this program is called into memory, this table of constants is also read in. The only reason a programmer would want to build a table this way is if the data in the items is never to change and if no items are to be inserted into or deleted from the table.

The items may be accessed either by item location or by key comparison. If the items are accessed by location, the data in the items should never change. This method of table search requires that the programmer know which item location is desired and use the TFINDD instruction to access the desired item.

If the items are accessed by key comparison rather than item location, the contents of these keys (and optionally the contents of the entire item) are built into the table with data definitions. At least the contents of the keys must remain the same from one processing day to the next; the other data in the items may either remain constant or change depending upon program requirements. However, the programmer must note that if he changes data in the table during one processing run, the next time this program is called into memory, the table contains the original data (that specified by the data definitions) and not the updated information. These items may be accessed in one of three ways:

- Use TFINDR if the items are to be accessed randomly and if the keys are not sequentially organized, or if it is a relatively small table (1-30 items) and positioning after a not-found search is not critical.
- Use TFINDB, TFINDS, or TFINDO if the items are to be accessed randomly and if the keys are sequentially organized.
- Use TFINDN or TFINDP if the items are to be accessed serially (one after the other).

### CONVENTIONS

The following rules apply to a fixed-length table that uses the slot processing technique (structure 1):

- The maximum length permitted for each key is 255 characters.
- The table instructions that can be executed on this table are:

  TBEGB, TBEGF
  TBILDD, TBILDN
  TDEL
  TFINDN, TFINDD, TFINDR, TFINDS, TFINDB, TFINDP, TFINDO
  TSHIFT
  TPACK
  TSORTA, TSORTD
  TMARK
  TRESET
  TJUMP

## FIXED—LENGTH TABLE WITHIN A RECORD

## USING THE SLOT PROCESSING TECHNIQUE

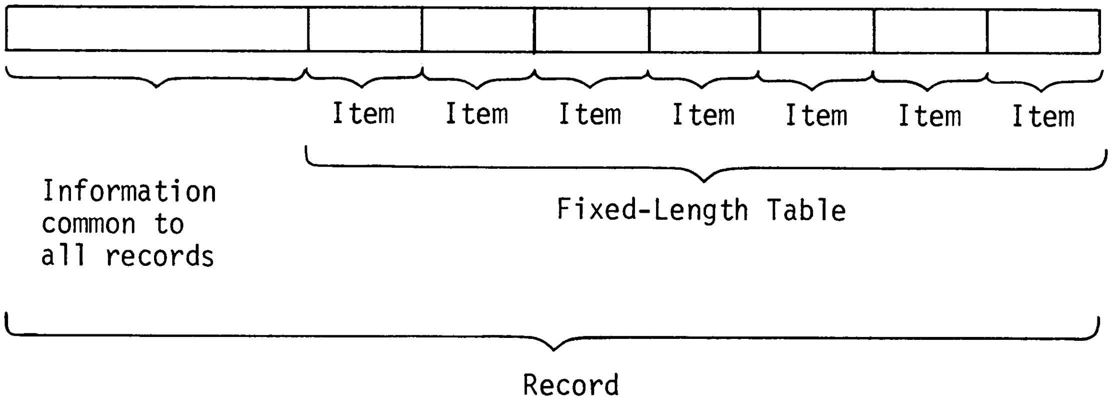
( STRUCTURE 2)

A table whose length never changes during processing is a fixed-length table. In other words, each item is the same length as the other items, and the overall length of the table remains the same during processing.

A fixed-length table may reside in each record in a file. The table may be preceded or followed by fields of data common to all records, e.g. customer name, address, account number, etc.

The following illustration shows the format of a fixed-length table within a record.

FORMAT OF A FIXED—LENGTH TABLE WITHIN A RECORD



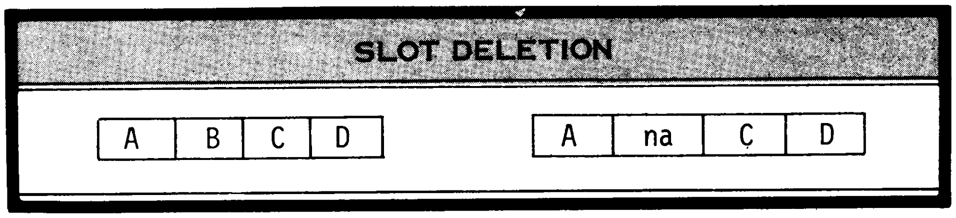
### SLOT PROCESSING TECHNIQUE

Slot processing is the only technique that can be used on fixed-length tables. Slot processing is a technique of table handling that allows items to be added to or deleted from the table not by altering the overall length of the table, but by simply considering each item to be either active or nonactive.

## To Delete an Item Using TDEL

To delete an item from the table, the TDEL instruction places a code in the item. It now considers this item to be nonactive or deleted.

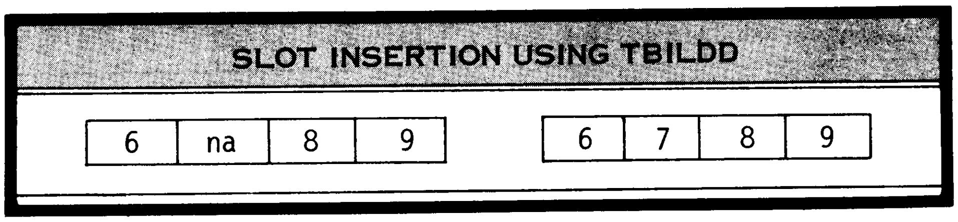
In the following example, TDEL deletes item B from the table using the slot processing technique. Note that the table size does not change.



## To Insert an Item Using TBILDD

To insert an item into a table, TBILDD directly inserts the item by placing a new item into the location specified by the user. However, if the location is currently active, TBILDD does not insert the new item but sets the E flag and transfers control to the routine specified by the branch operand.

In the following example, TBILDD is to insert a new item into the second item position. This new item has a key of 7.



## BUILDING THE TABLE

A fixed-length table within a record (structure 2) may be initially built in any of three ways: through the execution of TBILDD, through the execution of TBILDN, or through initial file input (data definitions).

The following guidelines are by no means definitive. They are merely suggested approaches to building a table. If a programmer has an application that will process well by combining some of the following suggestions, he is free to do so as long as he does not violate any conventions pertaining to table structure or instructional use, e.g. he may not in any circumstance use a TSERT instruction on a fixed-length table within a record.

### Building the Table with TBILDD

TBILDD is used to build a fixed-length table that is to be processed by item location and that has insertion/deletion capabilities.

Since all fixed-length tables within a record must use the slot processing technique, the only way items can be inserted into the table is through the execution of TBILDD. TBILDD does not look at the contents of the keys to determine where to build the item. It builds an item into the first, fourth, tenth, fiftieth, etc. item location in the table. This requires that the programmer know into which location the item is to be inserted.

Hence, if items are to be added to or deleted from the table, the programmer should process this table by item location rather than by key comparison. The following instructional variations logically complement each other and should be used when processing a table within a record that has insertion/deletion capabilities:

- To build the table, use TBILDD.
- To access items within the table, use TFINDD.
- To insert items into the table, use TBILDD.
- To delete items from the table, use TDEL.

## Building the Table with TBILDN

TBILDN is used to build during program execution a fixed-length table that is to be processed by key comparison but that has no insertion/deletion capabilities.

A programmer may wish to build during program execution a fixed-length table within each record in a file. The keys of each table are constant for today's run, i.e. the program will never insert items into or delete items from the table once the table is built. Tomorrow, however, an entirely new table may be built into each record whose keys may or may not resemble today's keys. (The data comprising the actual contents of the keys may originate from a COT reader or from magnetic media as an output of another program.)

The following instructional variations complement each other and should be used when processing a fixed-length table by key comparison:

- To build the table, use TBILDN.
- To access items within the table:
    - Use TFINDN or TFINDP if the items are to be accessed sequentially (one after the other).
    - Use TFINDR if the items are to be accessed randomly and if the keys are not sequentially organized.
    - Use TFINDB, TFINDS, or TFINDO if the items are to be accessed randomly and if the keys are sequentially organized.

## Building the Table Through Initial File Input (Data Definitions)

A fixed-length table within a record may be built through initial file input if no items are to be added to or deleted from the table and if the data (or keys) in the items are to remain constant from one processing day to the next. The only reason a programmer would want to build a table this way is if each record is to contain a table of unchanging data, i.e. if each record is to contain a table of constants.

The items may be accessed either by item location or by key comparison. If the items are accessed by location rather than by key comparison, the data in the items should never change. This method of table search requires that the programmer know which item location is desired and use the TFINDD instruction to access the desired item.

If the items are accessed by key comparison rather than item location, the contents of these keys (and optionally the contents of the entire item) are built into the record during initial file input, i.e. from the data deck. At least the contents of the keys must remain the same from one processing day to the next; the other data in the items may either remain constant or change depending upon program requirements. However, the programmer must note that if he changes data in the table during one processing run, the next time this file is processed, the table in each record contains the updated information and not the original data. These items may be accessed in one of three ways:

- Use TFINDR if the items are to be accessed randomly and if the keys are not sequentially organized, or if positioning after a not-found search is not critical.
- Use TFINDB or TFINDS if the items are to be accessed randomly and if the keys are sequentially organized.
- Use TFINDN if the items are to be accessed serially (one after the other).

## CONVENTIONS

The following rules apply to a fixed-length table that uses the slot processing technique (structure 2):

- The maximum length permitted for each key is 255 characters.
- The table instructions that can be executed on this table are:

  TBEGB, TBEGF
  TBILDD, TBILDN
  TDEL
  || TFINDN, TFINDD, TFINDR, TFINDS, TFINDB, TFINDP, TFINDO
  TSHIFT
  TPACK
  TSORTA, TSORTD
  TMARK
  TRESET
  TJUMP

# FIXED—LENGTH TABLE (MINOR TABLE) IN AN ITEM OF ANOTHER TABLE USING THE SLOT PROCESSING TECHNIQUE (STRUCTURE 3)

## DESCRIPTION

A major table, i.e. a table within an area or a table within a record, may contain a minor table which subdivides the data in each item in the major table.

Each item in the major table contains the keys of the major table, any optional fields of data that are needed, and a minor table. Consider the following illustration.

**MINOR TABLE WITHIN AN ITEM IN A MAJOR TABLE**

| Keys in Major Table | | Optional Data | Minor Table | | | | |
|---|---|---|---|---|---|---|---|
| Major Key | Minor Key | xxxxxxxxxxxxx | Item | Item | Item | Item | Item |

For instance, a major table may limit its items to a specific make and model of automobile, and the minor table may further break down each car into its parts and their prices.

The following example shows how this data could be arranged. The number of items in the minor table can be expanded to include many other parts and prices. This illustration only shows how live data would look in both the major and minor tables.

**DATA IN A MINOR TABLE**

| Keys in Major Table | Items in Minor Table | | |
|---|---|---|---|
| Make A Model A | Part X Price | Part Y Price | Part Z Price |
| Make A Model B | Part X Price | Part Y Price | Part Z Price |
| Make B Model A | Part X Price | Part Y Price | Part Z Price |

If the price of part Z in Make A Model B is desired, the program first searches the major table for Make A Model B. Once this item in the major table is made accessible, the program searches the minor table within the item for part Z. The desired item in the minor table is now accessible, and the program can access the price of part Z in the Make A Model B car.

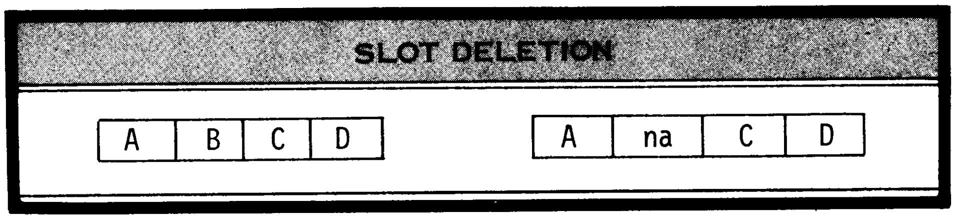
## SLOT PROCESSING TECHNIQUE

Slot processing is the only technique that can be used on minor tables. Slot processing is a technique of table handling which allows items to be added to or deleted from the table not by altering the overall length of the table, but by simply considering each item to be either active or nonactive.

### To Delete an Item Using TDEL

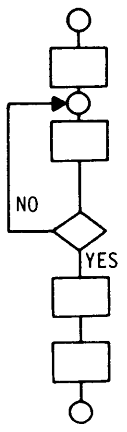
To delete an item from the table, the TDEL instruction places a code in the item. It now considers this item to be nonactive or deleted.

In the following example, TDEL deletes item B from the table using the slot processing technique. Note that the table size does not change.



### To Insert an Item Using TFINDN, COMP, and MOVE

To insert into a minor table, the TFINDN instruction is used to find the desired item location. The COMP instruction must be used as each item becomes accessible to make a key comparison. When the desired item location is found, the MOVE instruction must be used to insert the new data. Consider the following illustration.



| | |
|---|---|
| TBEGF | Initialize the minor table. |
| TFINDN | Select the first item in the table, when following TBEGF. When encountered subsequently select next sequential item. |
| COMP | Test for desired key. If NO, branch to FINDN; if YES, to MOVE. |
| MOVE | Insert data into item. |
| BR | To next appropriate instruction. |

In the following example, the second item location has been made accessible by the TFINDN and COMP instruction and the MOVE instruction inserts the new data with a key of 7.

**SLOT INSERTION USING TFINDN, COMP, MOVE**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | na | 8 | 9 | | 6 | 7 | 8 | 9 |

## BUILDING THE TABLE

Since the TBILD instructions cannot be used on a minor table, the programmer must construct all minor tables himself. The following program constructs a major and a minor table concurrently. The program requires a transaction file, a table area, and a workarea.

### Transaction File

A large firm wishes to take inventory on all pieces of business equipment in its office complex. It records the quantity of each type of equipment in each office on a transaction record and inputs these records to the program.

**TRANSACTION RECORDS**

| Coded Equipment Number | Floor | Office Number | Quantity in Office |
|---|---|---|---|

### Table Area

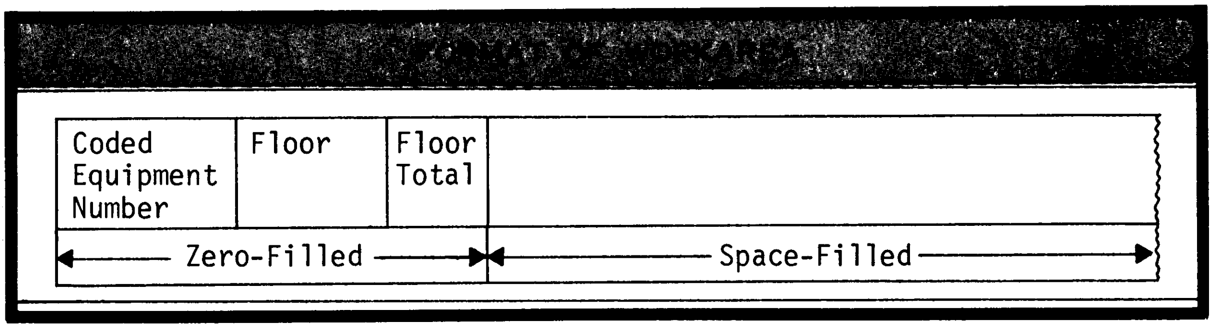
The program is to access a transaction record and use the information in it to build the major and the minor tables. The programmer reserves an area in memory for the entire table. Each item in this major table contains a minor table. The following illustration shows the format of each item in the major table.

**MINOR TABLE WITHIN AN ITEM IN THE MAJOR TABLE**

| Keys in Major Table | | | Items in Minor Table | | | |
|---|---|---|---|---|---|---|
| Coded Equipment Number | Floor | Floor Total | Office Number | Quantity in Office | Office Number | Quantity in Office |

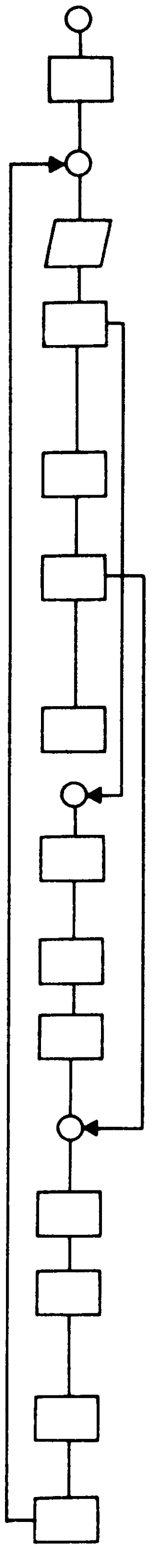
## Workarea

Before the program can build an item into the major table, it must first construct this item in a workarea. The first three fields of the workarea are initially zero-filled. The remainder of the area (that part corresponding to the items in the minor table) is space-filled.



## Procedural Instructions

The procedural instructions access a transaction record and use the information in it to build the major and the minor tables.



| | |
|---|---|
| TBEGB | Initialize the major table. |
| GET | A transaction record. |
| TFINDR | Search the major table for the item whose keys correspond to the equipment and floor fields in the transaction record. If the item does not exist in the major table, branch to build the item into the table. |
| ADD | The contents of the office field to the contents of the floor-total field. |
| TFINDR | Search the minor table within the accessed major item for the minor item whose keys correspond to the office field in the transaction record. If this item does not exist in the minor table, branch to build the item into the table. |
| BR | Branch to a TRANSERR routine. If control falls through to this instruction, the current transaction record is a duplicate of a previous one. |
| MOVE | The contents of the equipment and floor fields in the transaction record into the corresponding fields in the workarea. |
| MOVE | The contents of the office field to the contents of the floor-total field. |
| TBILDN | In the major table the item contained in the workarea. |
| TBEGF | Initialize the minor table. |
| TFINDR | Find the first ' Ø ' in the minor table. Since the table area is originally space-filled, this instruction accesses the first vacant item position in the minor table. |
| MOVE | The office and quantity fields from the transaction record into the corresponding fields in the accessed minor table. |
| BR | To get another transaction record. |

## CONVENTIONS

- The major table must be fixed in length.
- The minor table must be fixed in length.
- The minor table must use the slot processing technique.
- The maximum length permitted for each key in the minor table is 255 characters.
- The table instructions that can be executed on a minor table are:

  TBEGF
  TFINDN, TFINDD, TFINDR, TFINDS, TFINDB, TFINDP, TFINDO
  TDEL
  TSHIFT
  TPACK
  TSORTA, TSORTD
  TMARK
  TRESET
  TJUMP

- The minor table generally has an offset of at least the key length of the major table.

## SOURCE-LINE ORGANIZATION

The table specification sheet and the item and field definitions for the minor table must follow the item and field definitions for the major table.

## VARIABLE–LENGTH TABLE WITHIN AN AREA USING THE SLOT PROCESSING TECHNIQUE

(STRUCTURE 4)

### DESCRIPTION

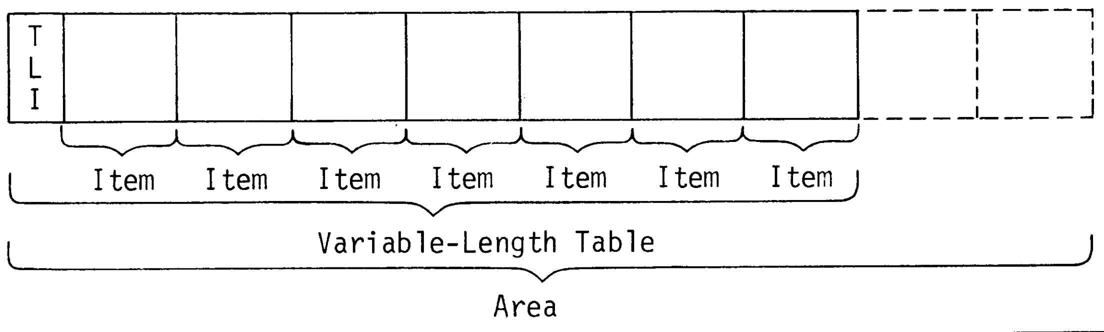
A table whose length may expand or contract during processing is a variable-length table. A variable-length table must contain fixed-length items, but the number of items in the table may increase or decrease during processing.

During processing, a programmer may frequently wish to access a variable-length table containing information common to many records or even to many files. To do this, he may define this table to be freestanding in a memory area.

Each variable-length table must have a table length indicator (TLI) as its first two 8-bit characters. This TLI must contain a binary number that specifies the total number of characters (including the two characters for the TLI) currently in the table.

A variable-length table expands or contracts by increasing or decreasing the total number of items in the table. In the following example, the table is variable in length. Note the table length indicator.

FORMAT OF A VARIABLE–LENGTH TABLE WITHIN A MEMORY AREA



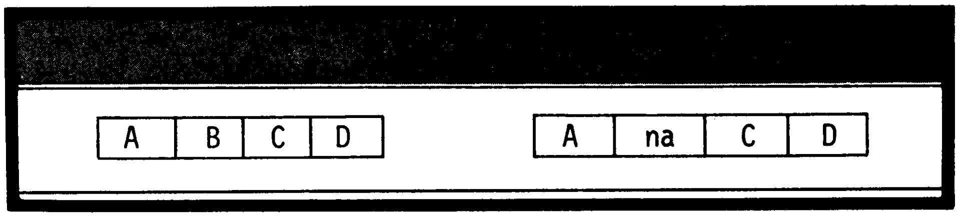
The programmer's definition of this area must reserve enough memory to contain the maximum size table. The compiler assigns an area in memory to this table from the programmer's entries on the data layout sheets.

## SLOT PROCESSING TECHNIQUE

Slot processing is a table-handling technique in which software considers items to be either active or nonactive. When the slot processing technique is used on a variable-length table within an area (structure 4), items may be deleted from the table with a TDEL instruction, and they may be inserted into the table with either a TBILDD or a TSERT instruction.

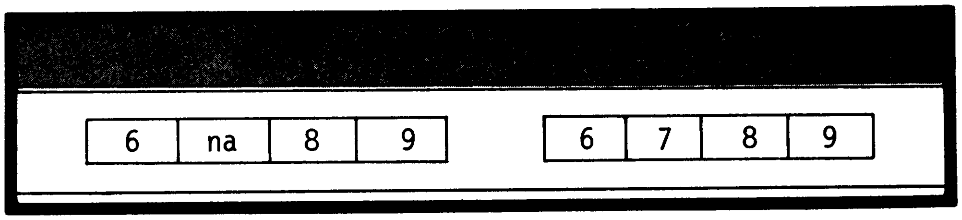
### To Delete an Item Using TDEL

To delete an item from the table, the TDEL instruction places a code in the item. It now considers this item to be nonactive or deleted. In the following example, TDEL deletes item B from the table using the slot processing technique. Note that the table size remains the same after deletion as it was before deletion.

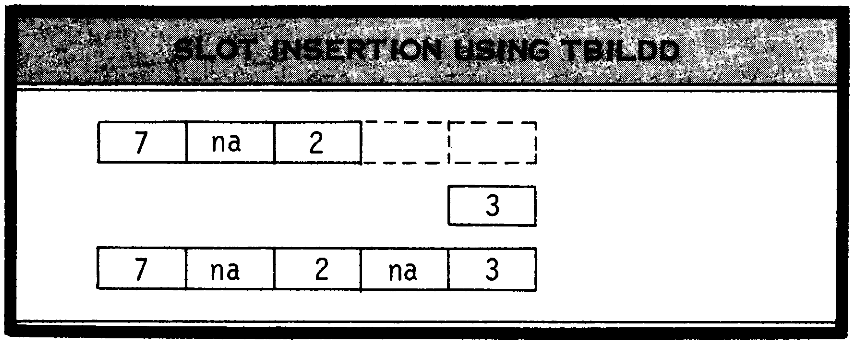


### To Insert an Item Using TBILDD

TBILDD directly inserts an item into the table by placing a new item into the table in the location specified by the user. However, if the location is currently active, TBILDD does not insert the new item but sets the E flag and transfers control to the routine specified by the branch operand.



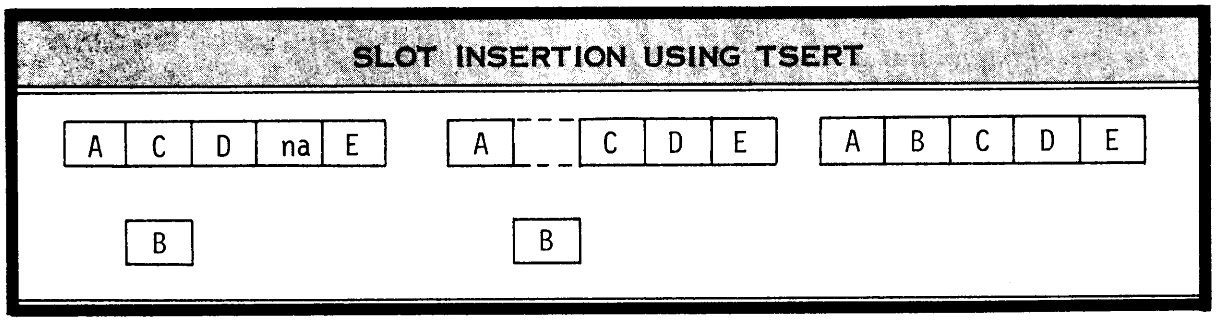
In the next illustration, TBILDD is to place a new item into the fifth item location of a variable-length table. Since the table currently contains only three items, TBILDD expands the length of the table to accommodate the insertion.



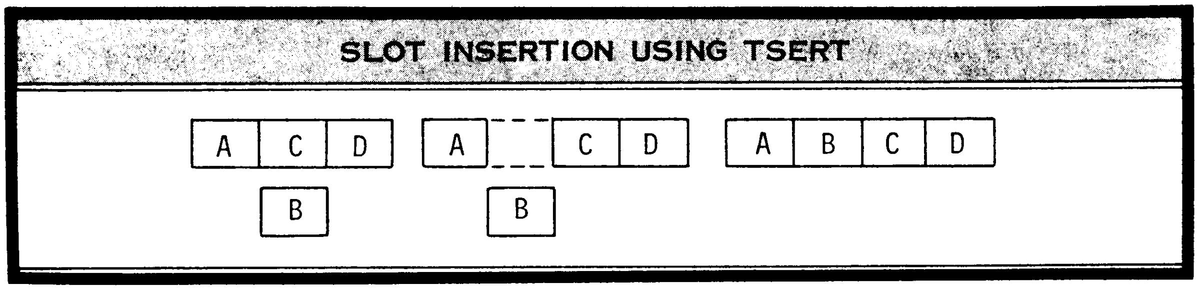
To Insert an Item Using TSERT

TSERT looks for the first nonactive item following the desired position, pushes down the intermediate active items in the table, and inserts the new item into its proper location.

In the following example, item B is to be inserted between items A and C. TSERT pushes items C and D down the table to fill up the first nonactive item encountered. It then inserts item B into the vacant position.



If, however, there is no nonactive item following the desired position, TSERT borrows a function of the pushdown technique and pushes one item length down the table all items following this position, thereby extending the table length by one item. It then inserts the new item into the position it has just vacated. Consider the following example.



## BUILDING THE TABLE

A variable-length table within an area using the slot processing technique (structure 4) may be initially built through the execution of either TBILDD or TBILDN.

The following guidelines are by no means definitive. They are merely suggested approaches to building a table. If a programmer has an application that will process well by combining some of the following suggestions, he is free to do so as long as he does not violate any conventions pertaining to table structure or instructional use.

### Building the Table with TBILDD

TBILDD is used to build a variable-length table that is to be processed by item location rather than by key comparison.

TBILDD does not look at the contents of the keys to determine where to build the item. It builds an item into the first, fourth, tenth, fiftieth, etc. item location in the table. This requires that the programmer know into which position the item is to be inserted.

Hence, if the programmer decides to process this table by item location rather than by key comparison, he should use the following instructional variations which logically complement each other:

- To build the table, use TBILDD.
- To access items within the table, use TFINDD.
- To insert items into the table, use TBILDD.
- To delete items from the table, use TDEL.

### Building the Table with TBILDN

TBILDN is used to build a variable-length table that is to be processed by key comparison rather than by item location.

A programmer may wish to build a variable-length table in a memory area. Later during processing the programmer can access the items by key comparison, store information in the items, obtain information from the items, and add items to and delete items from the table.

The following instructional variations complement each other and should be used when processing a variable-length table by key comparison:

- To build the table, use TBILDN.
- To access items within the table:
  - Use TFINDN or TFINDP if the items are to be accessed sequentially (one after the other).
  - Use TFINDR if the items are to be accessed randomly and if the keys are not sequentially organized.
  - Use TFINDB, TFINDS, or TFINDO if the items are to be accessed randomly and if the keys are sequentially organized.

- To insert items into the table, use TSERT.
- To delete items from the table, use TDEL.

## CONVENTIONS

The following rules apply to a variable-length table within an area using the slot processing technique (structure 4):

- The first two characters in the table must be the table length indicator (TLI).
- The maximum length permitted for each key is 255 characters.
- The programmer's definition of the area must reserve enough memory to contain the maximum size table.
- Extremely large tables with a high degree of insertion/deletion lend themselves to this type of table.
- All table instructions can be executed on this table. These are:

  TBEGB, TBEGF
  TBILDN, TBILDD
  TFINDN, TFINDD, TFINDR, TFINDS, TFINDB, TFINDP, TFINDO
  TSERT
  TDEL
  TSHIFT
  TSORTA, TSORTD
  TPACK
  TMARK
  TRESET
  TJUMP

## VARIABLE—LENGTH TABLE WITHIN AN AREA

## USING THE PUSHDOWN PROCESSING TECHNIQUE

( STRUCTURE 5 )

### DESCRIPTION

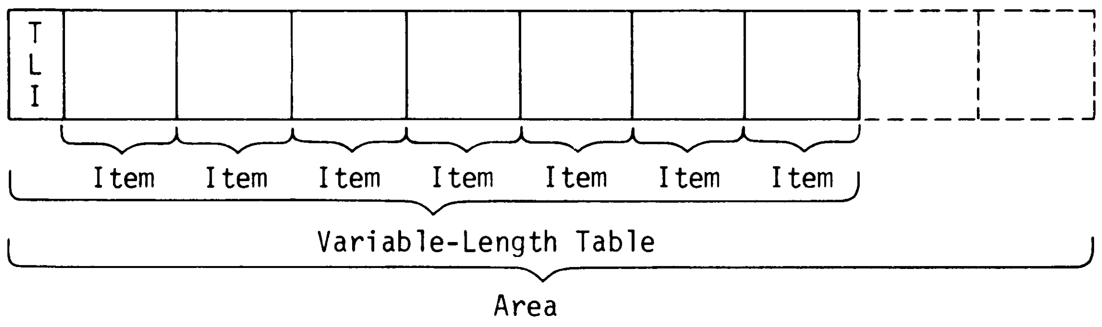
A table whose length may expand or contract during processing is a variable-length table. A variable-length table must contain fixed-length items, but the number of items in the table may increase or decrease during processing.

During processing, a programmer may frequently wish to access a variable-length table containing information common to many records or even to many files. To do this, he may define this table to be freestanding in a memory area.

Each variable-length table must have a table length indicator (TLI) as its first two 8-bit characters. This TLI must contain a binary number that specifies the total number of characters (including the two characters for the TLI) currently in the table.

A variable-length table expands or contracts by increasing or decreasing the total number of items in the table. In the following example, the table is variable in length. Note the table length indicator.

**FORMAT OF A VARIABLE—LENGTH TABLE WITHIN A MEMORY AREA**

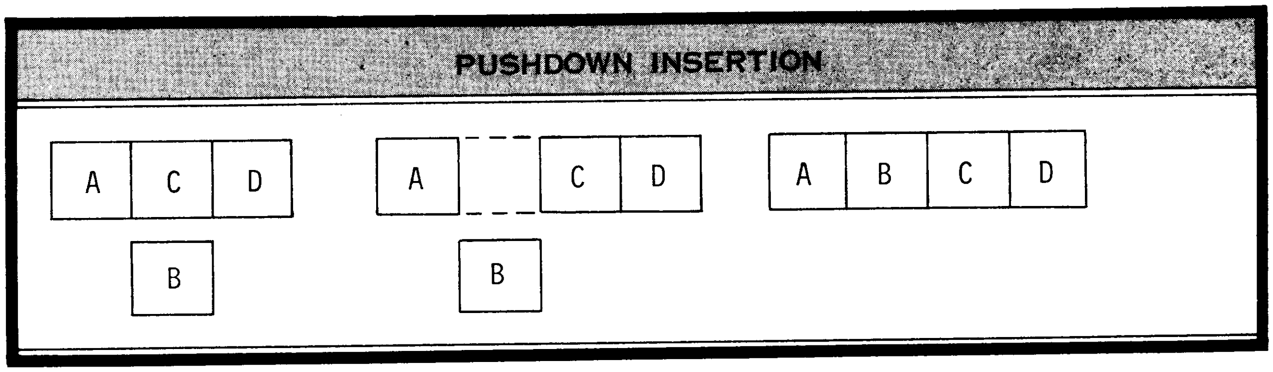


The programmer's definition of this area must reserve enough memory to contain the maximum size table. The compiler assigns an area in memory to this table from the programmer's entries on the data layout sheets.

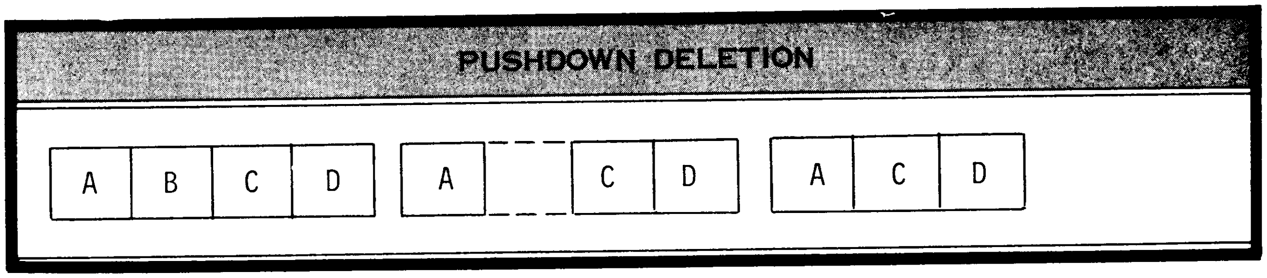
## PUSHDOWN PROCESSING TECHNIQUE

A variable-length table within a memory area uses the pushdown processing technique. Pushdown processing is a technique of table handling in which the size of a table expands and contracts as items are added to and deleted from the table.

Consider the following example. Before item B is inserted into its proper place, the TSERT instruction pushes down all items in the table that are to follow item B. This extends the length of the table as far as necessary to accommodate the insertion. TSERT then inserts item B.



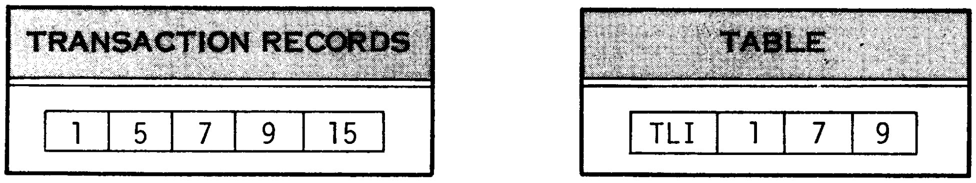
To delete an item, the reverse is true. The TDEL instruction removes item B from the table. Then, to fill the vacant position, it pushes up the table all items which followed item B, thereby contracting the length of the table. Consider the following illustration.

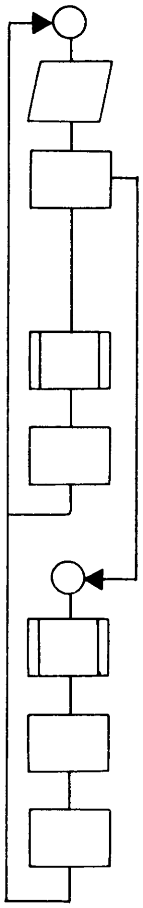


## SAMPLE PROGRAM

The following program illustrates the use of a variable-length table within an area. The table uses the pushdown processing technique and contains sequentially organized items.

The program is to access each transaction record and update its corresponding item in the table. If the desired item is not in the table, the program is to insert it into its proper location. The following procedural instructions perform these functions.





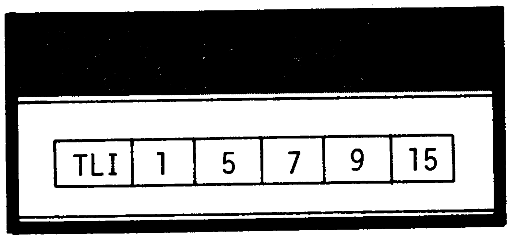
| | |
|---|---|
| GET | A transaction record. |
| TFINDS | In the table the item whose key corresponds to the key in the transaction record. If this item does not exist, transfer control to the routine specified by the branch operand. |
| Update | The item in the table. |
| BR | To get another transaction. |
| Construct | The new item in a workarea. |
| TSERT | The new item into the table. |
| BR | To get another transaction. |

Let's analyze the preceding program. When transactions 1, 7, and 9 are accessed, the items in the table corresponding to these transactions are updated.

When transaction 5 is accessed, TFINDS makes the item with key 7 accessible, sets the E flag, and transfers control to the routine specified by the branch operand. During the execution of this branch routine, TSERT pushes items 7 and 9 one item length down the table and inserts item 5 into the vacated position.

When transaction 15 is accessed, TFINDS makes accessible the position immediately following the item with key 9, sets the G flag, and transfers control to the routine referenced by the branch operand. During the execution of this branch routine, TSERT inserts item 15 immediately after item 9.

After all transactions have been processed, the table contains the updated items 1, 7, and 9 and the newly inserted items 5 and 15.



## BUILDING THE TABLE

Because of the very nature of a table using the pushdown processing technique, items are continually changing their location within the table. As an item is added to the table, all items that are to follow the item are pushed down the table to accommodate the insertion. As an item is deleted from the table, all items that followed this item are pushed up the table to fill the vacant position. Hence, it is virtually impossible to process by item location a variable-length table within an area that uses the pushdown processing technique.

All tables using the pushdown processing technique should be processed by key comparison. The table instructions are then able to access the desired item regardless of item location. Therefore, the TBILDD and TFINDD instructions are not used with this structure.

TBILDN is used to build a variable-length table that is to be processed by item location. Later during processing, the programmer can access the items by key comparison, store information in the items, obtain information from the items, and add items to and delete items from the table.

The following instructional variations logically complement each other and should be used when processing a table by key comparison:

- To build the table, use TBILDN.
- To access items in the table:
  - Use TFINDN or TFINDP if the items are to be accessed sequentially (one after the other).
  - Use TFINDR if the items are to be accessed randomly and if the keys are not sequentially organized, or if the table is relatively small (1-30 items) and positioning after a not-found search is not critical, i.e., TSERT is not required.
  - Use TFINDB, TFINDS, or TFINDO if the items are to be accessed randomly and if the keys are sequentially organized.
- To insert items into the table, use TSERT.
- To delete items from the table, use TDEL.

## CONVENTIONS

The following rules apply to a variable-length table within an area using the pushdown processing technique (structure 5):

- The first two characters in the table must be the table length indicator (TLI).
- The maximum length permitted for each key is 255 characters.
- The programmer's definition of the area must reserve enough memory to contain the maximum size table.
- The table instructions that can be executed on this table are:

  TBEGB, TBEGF
  TBILDN
  TFINDN, TFINDR, TFINDS, TFINDB, TFINDP, TFINDO
  TSERT
  TDEL
  TSHIFT
  TSORTA, TSORTD
  TMARK
  TRESET
  TJUMP

## VARIABLE–LENGTH TABLE WITHIN A RECORD

## USING THE PUSHDOWN PROCESSING TECHNIQUE

(STRUCTURE 6)

### DESCRIPTION

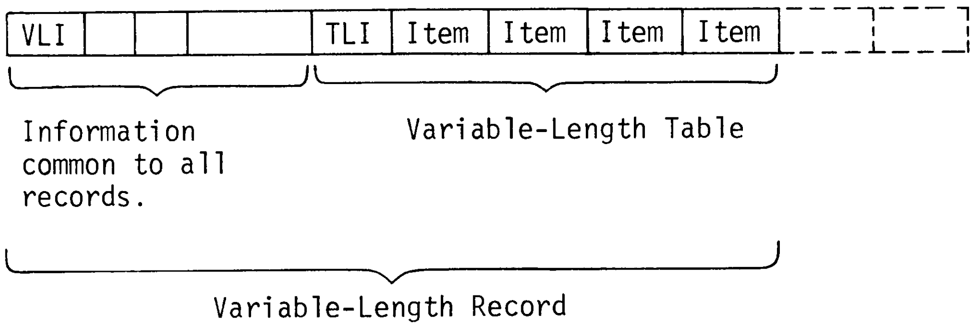
A table whose length may expand or contract during processing is a variable-length table. A variable-length table must contain fixed-length items, but the number of items in the table may increase or decrease during processing.

A variable-length table may reside in each record in a file. This table contains information that may have been systematically input during numerous processing runs. A variable-length table in a record must occupy the last portion of a variable-length record.

Each variable-length table must have a table length indicator (TLI) as its first two 8-bit characters. This TLI must contain a binary number that specifies the total number of characters (including the two characters for the TLI) currently in the table.

A variable-length table expands and contracts by increasing and decreasing the total number of items in the table. In the following example, the table is variable in length. Note the table length indicator.

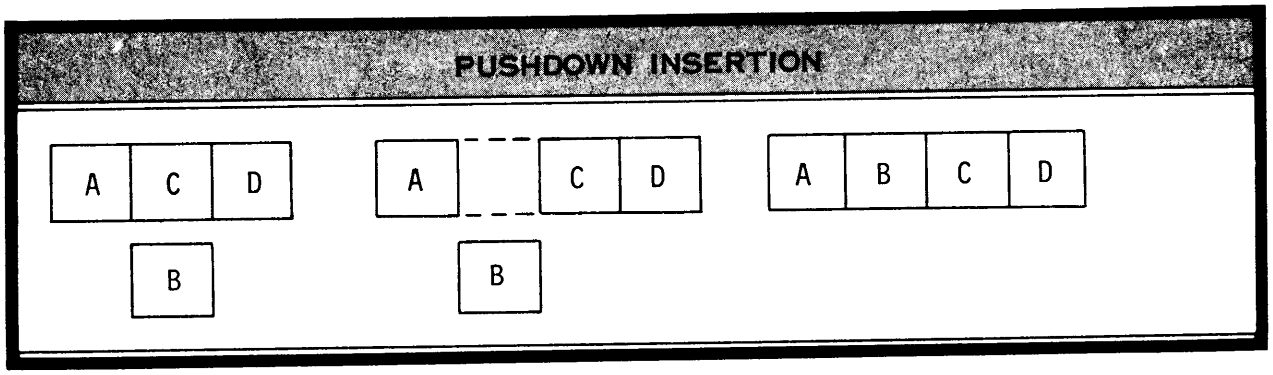
**FORMAT OF A VARIABLE–LENGTH TABLE WITHIN A RECORD**



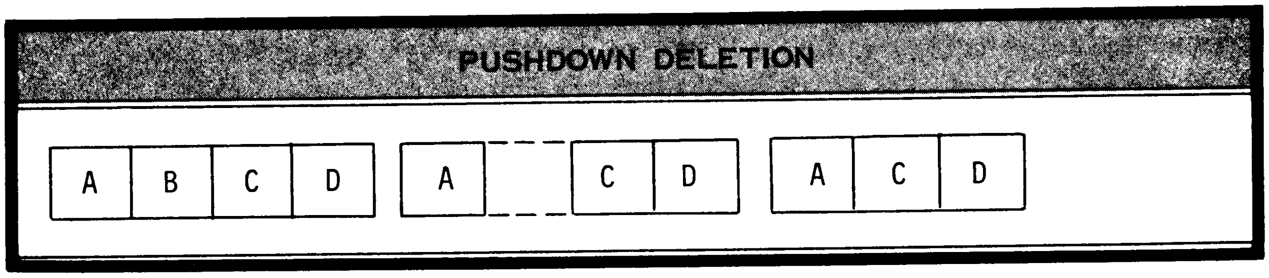
## PUSHDOWN PROCESSING TECHNIQUE

A variable-length table within a record uses the pushdown processing technique. Pushdown processing is a technique of table handling in which the size of a table expands and contracts as items are added to and deleted from the table.

Consider the following example. Before item B is inserted into its proper place, the TSERT instruction pushes down all items in the table that are to follow item B. This extends the length of the table as far as necessary to accommodate the insertion. TSERT then inserts item B.



To delete an item, the reverse is true. The TDEL instruction removes item B from the table. Then, to fill the vacant position, it pushes up the table all items which followed item B, thereby contracting the length of the table. Consider the following illustration.



## BUFFER-AREA CONSIDERATIONS

As the size of the table contracts and expands, the size of the record also contracts and expands. Therefore, before modifying the size of the table, use the DELETE instruction to move the current record out of the buffer area and into a workarea. This protects the information in other records in the input buffer area from being destroyed. Since the table is to be accessed while the record is in the workarea, two coding rules must be followed:

1. On data layout sheets define the workarea as containing all the fields that are to be accessed while the record is in the area. This includes the item and field definitions for the table.

2. On the table specification sheet (whose source line when input to the compiler must immediately precede the item definition), specify that this table is within a record. During processing, then, the table instructions treat this table as a table within a record.

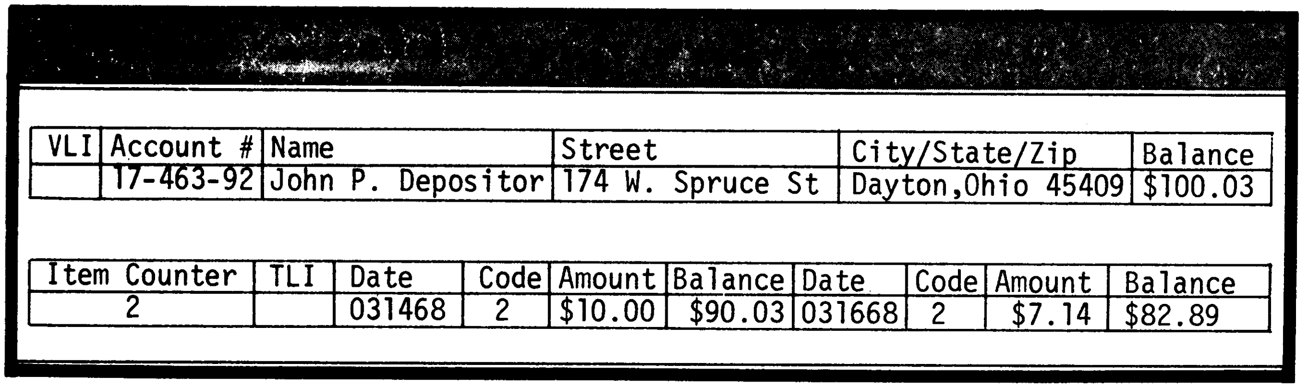
If a variable-length table within a record is to be accessed during a particular run that does not alter the length of the table or the record (for instance, if the program is to print each item in the master record), the record may remain within the input buffer area during processing. On data layout sheets, define the record as containing all the fields that are to be accessed while the record is in the buffer area. This includes the item and field definitions for the table.

## SAMPLE PROGRAM

A bank wishes to save each depositor's transactions within the depositor's master record. Later the bank can copy these transactions (withdrawals and deposits) onto the monthly statement sent to the depositor. The procedural instructions which perform this function require data definitions of the master file, the transaction file, and two workareas.

### Master File

The master file contains a record for each depositor. The record contains a variable-length table. Each time a depositor makes a transaction, a record of this transaction is kept in the master record. The following master record contains two tabled items.



| VLI | Account # | Name | Street | City/State/Zip | Balance |
|---|---|---|---|---|---|
| | 17-463-92 | John P. Depositor | 174 W. Spruce St | Dayton,Ohio 45409 | $100.03 |

| Item Counter | TLI | Date | Code | Amount | Balance | Date | Code | Amount | Balance |
|---|---|---|---|---|---|---|---|---|---|
| 2 | | 031468 | 2 | $10.00 | $90.03 | 031668 | 2 | $7.14 | $82.89 |

Note that the first field of the master record is the variable-length indicator (VLI) for the master record. The next five fields contain information common to all master records, i.e. information about the depositor and his account.

Note also that the seventh field in the record, the item counter, indicates that two items are currently contained in the table. The eighth field is the table length indicator (TLI) for the table.

Next comes the first item in the table. Each item (two in all) contains four fields: date, code, amount, and balance. (A code of 1 indicates that the transaction is a deposit; a code of 2 indicates that it is a withdrawal.)

## Transaction File

Each time a depositor makes a deposit or a withdrawal, the information is put onto a transaction record.

**FORMAT OF TRANSREC**

| ACCNO | NAME | DATE | CODE | AMOUNT |
|---|---|---|---|---|
| 17-463-92 | John P. Depositor | 031768 | 1 | $75.98 |

This transaction record is used to update the table in the master record.

## Workarea: NEWITEM

The item to be inserted into the table must first be constructed in a workarea.

**FORMAT OF NEWITEM**

| DATE | CODE | AMOUNT | BALANCE |
|---|---|---|---|

## Workarea: UPDATEMAS

Because the length of the master record is to be altered, the program moves the master record into a workarea called UPDATEMAS. UPDATEMAS is of the same format as is MASTEREC.

## Procedural Instructions

From information in the transaction record and in the last item of the table in the master record, the program constructs in the workarea NEWITEM the item to be inserted into the table. The program then builds into its proper place the item to be inserted into the table.

The table instruction which builds the new item automatically updates the information in the item counter and in the variable and table length indicators. After the master record has been updated, the program inserts the record back into its buffer area.

The updated master record contains the following information.

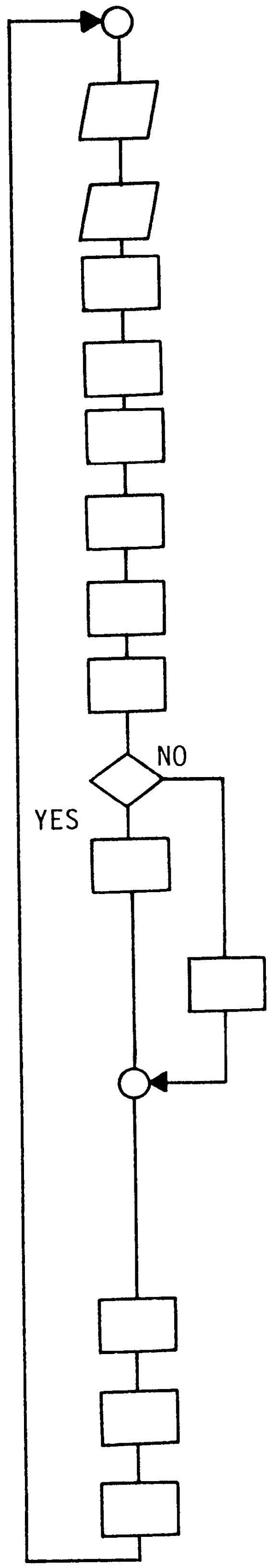
**UPDATED MASTER RECORD**

| VLI | Account # | Name | Street | City/State/Zip | Balance |
|---|---|---|---|---|---|
| | 17-463-92 | John P. Depositor | 174 W. Spruce St | Dayton, Ohio 45409 | $100.03 |

| Item Counter | TLI | Date | Code | Amount | Balance | Date | Code | Amount | Balance |
|---|---|---|---|---|---|---|---|---|---|
| 2 | | 031468 | 2 | $10.00 | $90.03 | 031668 | 2 | $7.14 | $82.89 |

| Date | Code | Amount | Balance |
|---|---|---|---|
| 031768 | 1 | $75.98 | $158.87 |

The steps needed to complete this problem are illustrated in the following flowchart.



| | |
|---|---|
| GET | A transaction record. |
| GET | The corresponding master record. |
| DELETE MASTEREC | from its buffer area, and place it in the UPDATEMAS area. |
| MOVE | The date in the transaction record into the date field in NEWITEM. |
| MOVE | The code in the transaction record into the code field in NEWITEM. |
| MOVE | The amount in the transaction record into the amount field in NEWITEM. |
| TBEGF | Initialize the table. |
| TFINDN | Find the last item in the table. |
| COMP | Is the transaction record a deposit? |
| ADD | The deposit to the balance field in the last item, and place the results in the balance field in NEWITEM. |
| SUB | The withdrawal from the balance field in the last item, and place the results in the balance field in NEWITEM. |
| TBILDN | Build this NEWITEM into the table at the next item location. |
| INSERT | The updated master record found in UPDATEMAS back into its buffer area. |
| BR | To get another transaction record. |

## BUILDING THE TABLE

Because of the very nature of a table using the pushdown processing technique, items are continually changing their location within the table. As an item is added to the table, all items that are to follow the item are pushed down the table to accommodate the insertion. As an item is deleted from the table, all items that followed this item are pushed up the table to fill up the vacant position. Hence, it is virtually impossible to process by item location a variable-length table within a record that uses the pushdown processing technique (structure 6).

All tables using the pushdown processing technique should be processed by key comparison. The table instructions are then able to access the desired item regardless of item location. Therefore, the TBILDD and TFINDD instructions are not used with this structure.

TBILDN is used to build a variable-length table that is to be processed by item location. Later during processing, the programmer can access the items by key comparison, store information in the items, obtain information from the items, and add items to and delete items from the table.

The following instructional variations logically complement each other and should be used when processing a table by key comparison:

- To build the table, use TBILDN.
- To access items in the table:
  - Use TFINDN or TFINDP if the items are to be accessed sequentially (one after the other).
  - Use TFINDR if the items are to be accessed randomly and if the keys are not sequentially organized, or if positioning after a not-found search is not critical, i.e., TSERT is not required.
  - Use TFINDB, TFINDS, or TFINDO if the items are to be accessed randomly and if the keys are sequentially organized.
- To insert items into the table, use TSERT.
- To delete items from the table, use TDEL.

## CONVENTIONS

The following rules apply to a variable-length table within a record using the pushdown processing technique (structure 6):

- The first two characters in the table must be the table length indicator (TLI).
- The maximum length permitted for each key is 255 characters.
- The table must occupy the last portion of a variable-length record.
- Before modifying the size of the table, the current record must be moved out of the buffer area and into a workarea. (See BUFFER-AREA CONSIDERATIONS in this publication.)
- The table instructions that can be executed on this table are:

  TBEGB, TBEGF
  TBILDN
  TFINDN, TFINDR, TFINDS, TFINDB, TFINDP, TFINDO
  TSERT
  TDEL
  TSHIFT
  TSORTA, TSORTD
  TMARK
  TRESET
  TJUMP

## TBEG INSTRUCTIONS

There are two variations of the TBEG instruction, TBEGB and TBEGF. One of the variations of TBEG must be the first table instruction encountered during the processing of a table. Each is discussed separately.

### TBEGB

#### Function

TBEGB (begin to build) initializes the building function of table processing. It assumes that a table is to be built or that an already existing table is to be completely rebuilt. Its execution ensures that the next table instruction encountered during processing starts its manipulations at the beginning of this table.

If the specified table uses the slot processing technique, TBEGB makes all the items in the table nonactive.

If the specified table is variable in length, TBEGB sets the TLI to 02, indicating that the table contains only the TLI.

If the specified table has the optional item counter, TBEGB sets it to zero.

If the table is within a record, TBEGB adjusts the VLI if necessary.

After TBEGB has been executed, the next table instruction encountered begins its table manipulations at the first item position in the table.

Consider the following illustration of a table initialized by TBEGB.

**TABLEAREA**

| ITEMCOUNTER | TLI | 1ST ITEM | 2ND ITEM | 3RD ITEM |
|---|---|---|---|---|
| 00 | 02 | nonactive | nonactive | nonactive |

Example

| OPERATION<br>18 19 20 21 22 23 | OPERANDS<br>24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
|---|---|
| TBEGB | TABLEA |

Conventions

Since it initializes the building function, TBEGB must be the first table instruction encountered during the processing of a table that is to be built.

If the table to be built is in a record, TBEGB should initialize the table in each record. One possible way to do this is to execute TBEGB after the GET. Consider the following example.

| OPERATION<br>18 19 20 21 22 23 | OPERANDS<br>24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
|---|---|
| GET | MASTEREC |
| TBEGB | MASTERTBL |

TBEGB may be reexecuted in the same program as many times as the programmer wishes to reinitialize the table.

TBEGB may be executed on the following tables:

- Fixed-length table within an area using the slot processing technique (structure 1).
- Fixed-length table within a record using the slot processing technique (structure 2).
- Variable-length table within an area using the slot processing technique (structure 4).
- Variable-length table within an area using the pushdown processing technique (structure 5).
- Variable-length table within a record using the pushdown processing technique (structure 6).

NOTE

Literal operands are not used with TBEGB.

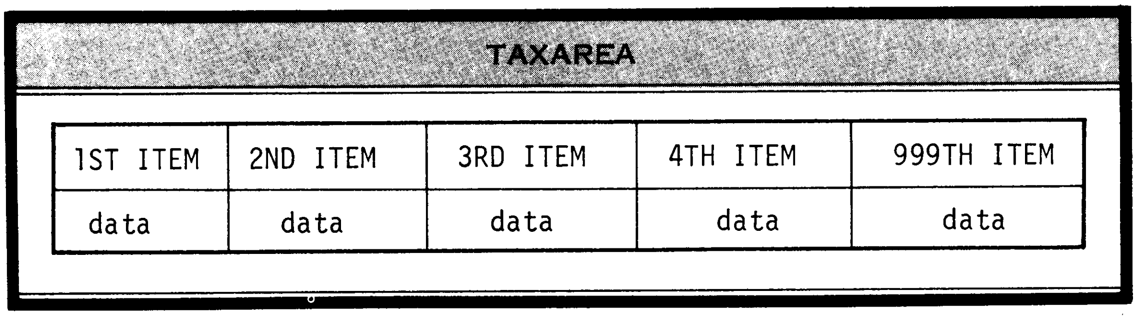
## TBEGF

### Function

TBEGF (begin to find) initializes all the functions of table processing except the building function. It assumes that a table already exists and that the table instructions following it are to access items, to insert and delete items, and/or to perform special table functions.

After execution of TBEGF, the next table instruction encountered begins its table manipulations at the first item position in the table.

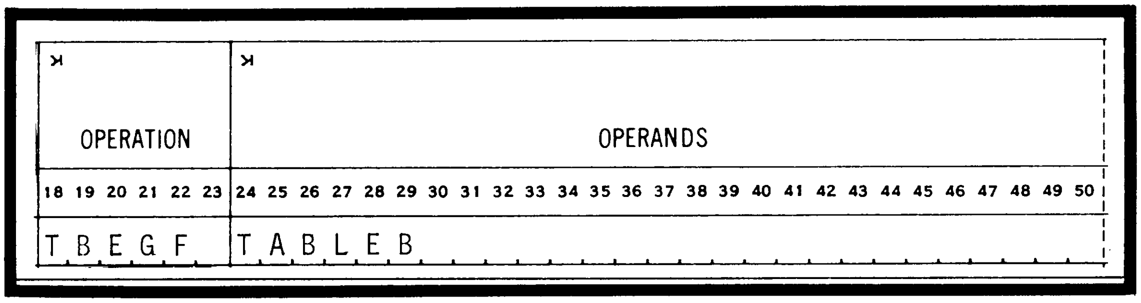
Consider the following illustration of a table initialized by TBEGF.



TAXAREA

| 1ST ITEM | 2ND ITEM | 3RD ITEM | 4TH ITEM | 999TH ITEM |
|---|---|---|---|---|
| data | data | data | data | data |

### Example



| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TBEGF | TABLEB |

### Conventions

Since it initializes all functions except the building function, TBEGF must be the first table instruction encountered during the processing of a table that has already been built.

If the table is in a record, TBEGF should initialize the table in each record. One possible way to do this is to execute the TBEGF after the GET. Consider the following example.

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| GET | MASTEREC |
| TBEGF | MASTERTBL |

TBEGF may be reexecuted in the same program as many times as the programmer wishes to reinitialize the table.

TBEGF may be executed on all types of tables.

NOTE

Literal operands are not used with TBEGF.

## TBILD INSTRUCTIONS

There are two variations of the TBILD instruction, TBILDN and TBILDD. Each is discussed separately.

### TBILDN

#### Function

TBILDN (build next) sequentially builds a table by placing an item into the next location in the table. It assumes that the table has previously been initialized.

If TBILDN is the first table instruction encountered after the execution of a TBEGB, TBILDN moves the item specified by the second operand into the first item position in the table.

If the execution of TBILDN is preceded by the execution of any table instruction (even another TBILDN) except TBEGB, TBILDN selects the next location in the table and then moves the item specified by the second operand into this location.

If the current table length is the maximum length allowed, the execution of TBILDN cannot build another item. Instead, it transfers control to the routine specified by the branch operand. The last item in the table is accessible to the program. The item specified by the second operand has not been built into the table.

If the specified table has the optional item counter, TBILDN increments it each time it builds an item into the table.

If the table is variable in length, TBILDN increments the TLI each time it builds an item into the table.

If the table is within a variable-length record, TBILDN increments the VLI each time it builds an item into the table.

#### Example

| OPERATION | OPERANDS | * |
|---|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 | 51 52 53 54 |
| TBILDN | TABLEA,ITEM5,TOOFAR | |

In positions 18 - 23, enter TBILDN.

As the first operand, enter the reference of the table to be built or accessed, TABLEA in this example.

As the second operand, enter the reference of the field containing the item, ITEM5 in this example, to be moved into the table.

As the branch operand, enter the reference of the user's routine to receive control, such as TOOFAR, when TBILDN tries to access a location beyond the limits of the table.

NOTE

> References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

Conventions

TBILDN may be used to construct either a fixed- or a variable-length table.

A fixed-length table within a record may initially be built using TBILDN. However, since the only technique permitted for updating these records is the slot processing technique using TBILDD, the programmer may wish also to build the table using TBILDD.

TBILDN may be used to construct the following tables:

- Fixed-length table within an area using the slot processing technique (structure 1).
- Fixed-length table within a record using the slot processing technique (structure 2).
- Variable-length table within an area using the slot processing technique (structure 4).
- Variable-length table within an area using the pushdown processing technique (structure 5).
- Variable-length table within a record using the pushdown processing technique (structure 6).

Literals are allowed in the second operand of TBILDN. If used, their length will be that of the image shown. X is the assumed type.

References used in the second operand of this instruction must not be contained within the table referenced by the first operand.

## TBILDD

### Function

TBILDD (build direct) has two functions: to build a table that has been initialized with TBEGB, and to insert an item into an existing table that has been initialized with TBEGF.

TBILDD assumes that the table uses the slot processing technique. It considers the first item in the table as item 1, the second item as item 2, etc. Each time TBILDD is executed, it either builds a table or inserts information specified by the second operand into an existing table at the item specified by the third operand.

If the specified item is beyond the current length of a variable-length table, TBILDD expands the table length to accommodate the insertion.

If the specified item is beyond the maximum length of the table, TBILDD sets the greater (G) flag and transfers control to the user's routine specified by the branch operand. The new item has not been inserted into the table.

If the specified item is currently active, TBILDD makes this active item accessible, sets the equal (E) flag, and transfers control to the user's routine specified by the branch operand. The new item has not been inserted into the table.

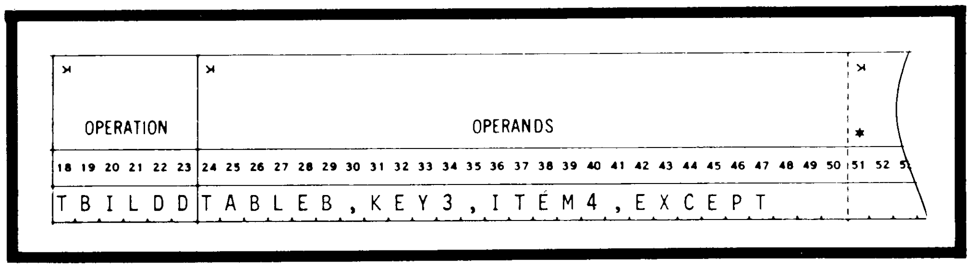
If the specified table has the optional item counter, TBILDD increments it each time it builds a table or inserts an item into the table.

If the table is variable in length, TBILDD increments the TLI each time it extends the length of the table.

TBILDD is to insert this item $\boxed{\text{Key 3}}$ into item 3 in each of the following tables.

| TABLES BEFORE TBILDD | | | TABLES AFTER TBILDD | | |
|---|---|---|---|---|---|
| Key 4 | Key 7 | Key 8 | Key 4 | Key 7 | Key 8 |
| Key 1 | Key 2 | Nonactive | Key 1 | Key 2 | Key 3 |
| Key 1 | | | Key 1 | Nonactive | Key 3 |

Example



| OPERATION | OPERANDS | * |
|---|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 | 51 52 |
| TBILDD | TABLEB,KEY3,ITEM4,EXCEPT | |

In positions 18 - 23, enter TBILDD.

As the first operand, enter the reference of the table to be built or accessed (TABLEB in this example).

As the second operand, enter the reference of the field containing the information (KEY3 in this example) to be inserted into the table.

As the third operand, enter the reference of the field containing an unsigned decimal number (ITEM4 in this example) that indicates which item in the table is to be built.

As the branch operand, enter the reference of the user's routine to receive control (EXCEPT in this example) when TBILDD tries to access an item beyond the limits of the table or when TBILDD accesses a currently active item.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

Conventions

TBILDD may be used to construct or to insert an item into the following tables:

- Fixed-length table within an area using the slot processing technique (structure 1).
- Fixed-length table within a record using the slot processing technique (structure 2).
- Variable-length table within an area using the slot processing technique (structure 4).

Literals are allowed in both the second and third operands of TBILDD. If used, their length will be that of the image shown. X is the assumed type of the second operand, and U is the assumed type of the third operand. The value of the third operand must never be zero.

References used in the second and third operands of this instruction must not be contained within the table referenced by the first operand.

## TFIND INSTRUCTIONS

There are seven variations of the TFIND instruction: TFINDN, TFINDD, TFINDR, TFINDS, TFINDB, TFINDP, and TFINDO. Each is discussed separately.

### TFINDN

#### Function

TFINDN assumes that TBEGF has previously been executed.

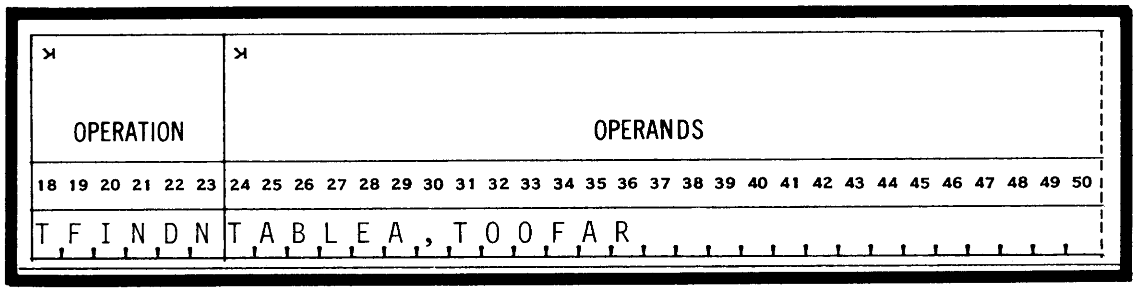
TFINDN selects the next sequential item in the table except in two instances:

- If the last table instruction encountered was a TBEGF, TFINDN selects the first item in the table.
- If the table uses the slot processing technique, TFINDN selects the next active item in sequence.

The data in the item selected by TFINDN can be accessed by the program.

If the currently accessible item is the last item in the table, the execution of TFINDN cannot select another item. TFINDN makes the last item in the table accessible to the program and transfers control to the routine specified by the branch operand.

#### Example



| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TFINDN | TABLEA,TOOFAR |

In positions 18 - 23 enter TFINDN.

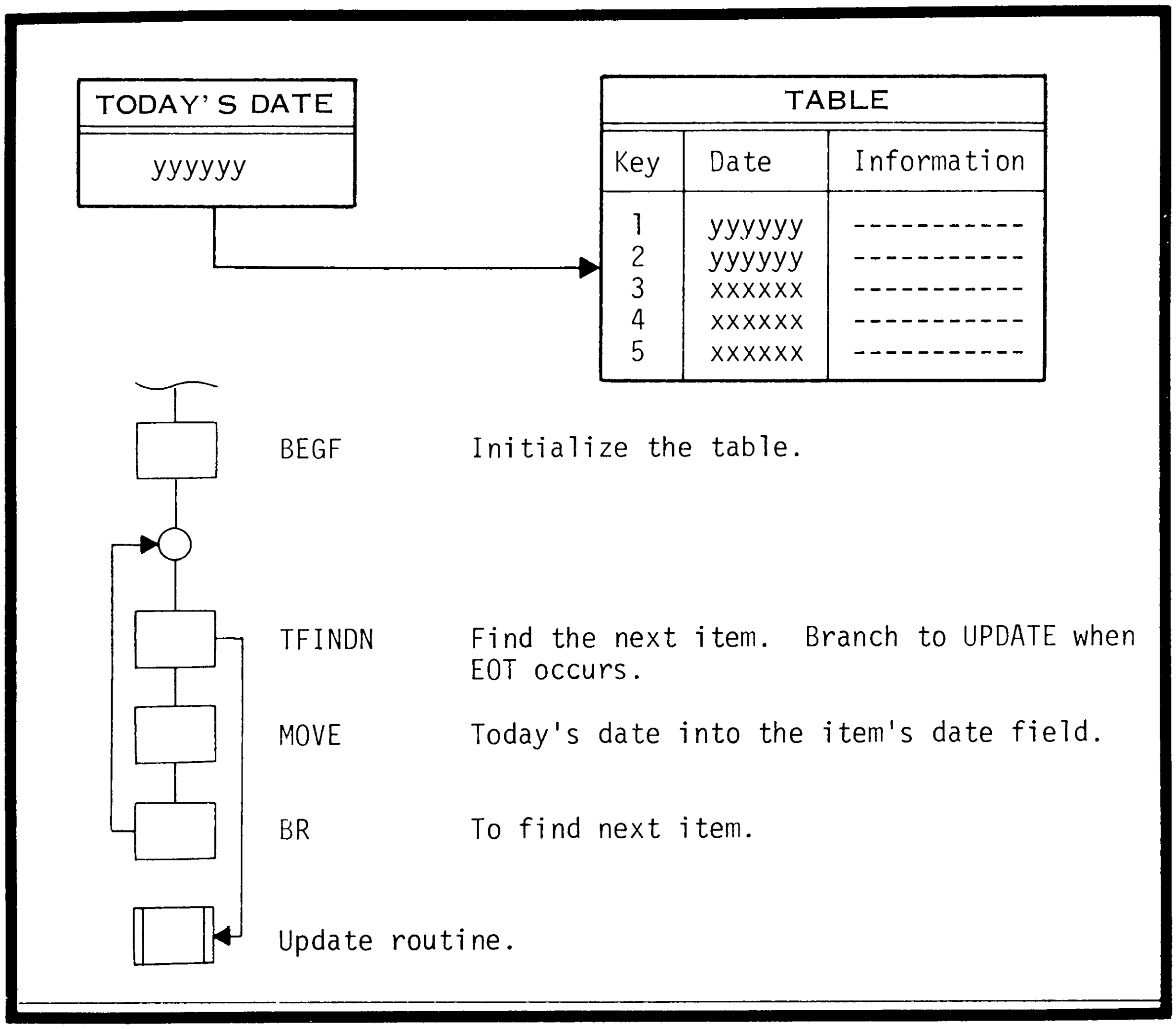
As the first operand, enter the reference of the table to be accessed, TABLEA in this example.

As the branch operand, enter the reference of the user's routine to receive control, such as TOOFAR, if the item accessed is the last item in the table.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

One possible use of TFINDN is to move today's date into each item in the table. Consider the following illustration:



## Conventions

TFINDN can be executed on all types of tables.

Literal operands are not permitted in TFINDN.

## TFINDD

### Function

TFINDD assumes that TBEGF has previously been executed. The items in the table may be arranged in ascending, descending, or random sequence of keys.

TFINDD uses the direct mode to select a specific item in the table by considering the first location in the table as item 1, the second location as item 2, etc. It accesses the table item which is specified by the second operand.

If the number of the specified item is greater than the number of the last item in the table, TFINDD transfers control to the routine specified by the branch operand.

If the table uses the slot processing technique and if the specified item is nonactive, TFINDD makes this nonactive item accessible, sets the equal (E) flag, and transfers control to the routine specified by the branch operand.

### Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TFINDD | TABLEB,ITEM6,TOOFAR |

In positions 18 - 23, enter TFINDD.

As the first operand, enter the reference of the table to be accessed, TABLEB in this example.

As the second operand, enter the reference of the field containing an unsigned decimal number, ITEM6 in this example, which indicates the desired item.

As the branch operand, enter the reference of the user's routine to receive control, such as TOOFAR, if the specified item is beyond the range of the table or if the specified item is nonactive.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

In the following illustration, TFINDD directly accesses the third item in the table.

| ITEM NO. | TABLE A | CONTENTS OF KEY | ACTION |
|---|---|---|---|
| 1 | Key / Data | 3 | |
| 2 | Key / Data | 1 | |
| *3 | Key / Data | 9 | Make this item accessible. |
| | | | |

In the following illustration, TFINDD is to directly access the sixth item in the table. If there are not six items in the table, TFINDD is to branch to a routine referenced by TOOFAR.

| ITEM NO. | TABLE B | CONTENTS OF KEY | ACTION |
|---|---|---|---|
| 1 | Key / Data | 9 | |
| 2 | Key / Data | 4 | |
| 3 | Key / Data | 10 | |
| 4 | Key / Data | 8 | |
| 5 | Key / Data | 1 | |
| *end of table | | | Branch to TOOFAR. |
| | | | |

Conventions

The items in the table may be arranged in ascending, descending, or random sequence of keys.

TFINDD may be executed on the following tables:

- Fixed-length table within an area using slot processing technique (structure 1).
- Fixed-length table within a record using slot processing technique (structure 2).
- Fixed-length table (minor table) within a table using slot processing technique (structure 3).

- Variable-length table within an area using slot processing technique (structure 4).

Literals are allowed in the second operand of TFINDD. If used, their length will be that of the image shown; i.e., a literal of 666 must be expressed as '666'. U is the assumed type. The value of the second operand must never be zero.

References used in the second operand of this instruction must not be contained within the table referenced by the first operand.

## TFINDR

### Function

TFINDR assumes that TBEGF has previously been executed. The items in the table may be arranged in ascending, descending, or random sequence of keys.

TFINDR performs a serial search for the desired item. It begins its search with the first item in the table and compares the key of this item with the contents of the second (and optionally third) operand.

If the compared fields are not equal, TFINDR selects the next item in sequence and compares its key to the contents of the second (and optionally third) operand.

If the compared fields are equal, a hit occurs. TFINDR makes the data in this item accessible to the user's program.

If the items in the table are exhausted before a hit occurs, TFINDR transfers control to the user's routine specified by the branch operand.

If the slot processing technique is used on this table, TFINDR ignores each nonactive item it encounters and selects the next item in sequence.

### Example

| OPERATION | OPERANDS | * |
|---|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 | 51 52 |
| TFINDR | TABLEB,'036250',,NOTFOUND | |
| TFINDR | TABLEB,'036250','76',NOTFOU | ND |

In positions 18 - 23, enter TFINDR.

As the first operand, enter the reference of the table to be accessed (TABLEB in this example).

As the second operand, enter the reference of the field containing the data (a literal of 036250 in this example) to be compared to the major key of each item in the table.

As the third operand (optional), enter the reference of the field containing the data (a literal of 76 in this example) to be compared to the minor key of each item in the table. If a 1-key format is used, enter a comma to indicate that no minor key comparison is made.

As the branch operand, enter the reference of the user's routine to receive control (such as NOTFOUND) if the compared fields are not equal.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

In the following illustration, TFINDR searches the table to make accessible the item with a key of 4. If the item is not in the table, TFINDR is to transfer control to the routine referenced by NOTFOUND. An asterisk indicates the first item that TFINDR selects.

| TABLE B | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key Data | 6 | 6 is not equal to 4. | Select next item. |
| Key Data | 3 | 3 is not equal to 4. | Select next item. |
| Key Data | 1 | 1 is not equal to 4. | Select next item. |
| Key Data | nonactive | | Select next item. |
| Key Data | 9 | 9 is not equal to 4. | Select next item. |
| *end of table | | | Branch to NOTFOUND. |

In the following illustration TFINDR is to make accessible the item with a key of 4. An asterisk indicates the first item that TFINDR selects.

| TABLE A | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key Data | 7 | 7 is not equal to 4. | Select next item. |
| Key Data | 2 | 2 is not equal to 4. | Select next item. |
| Key Data | 5 | 5 is not equal to 4. | Select next item. |
| Key Data | 9 | 9 is not equal to 4. | Select next item. |
| Key Data | 4 | 4 is equal to 4. | Make this item accessible. |

## Conventions

The items in the table may be in either ascending, descending, or random sequence of keys.

If the table has a small number of items, a serial search (TFINDR) proves quickest in accessing the individual items.

TFINDR may be executed on all types of tables.

Literals are allowed in both the second and third operands of TFINDR. If used, their length will be that of the image shown, i.e. a literal of ABC must be expressed as 'ABC'. X is the assumed type.

References used in the second and third operands of this instruction must not be contained within the table referenced by the first operand.

## TFINDS

### Function

TFINDS assumes that TBEGF has previously been executed. The items in the table must be arranged in either ascending or descending sequence of keys.

TFINDS performs a sequential search for the desired item. It begins its search with the first item in the table and compares the key of this item with the key of the desired item (the contents of the second operand). (Assume the keys to be in ascending sequence for documentary purposes.)

If the key of the item selected in the table is less than the key of the desired item, TFINDS selects the next item in sequence and compares its key with the key of the desired item.

If the compared fields are equal, a hit occurs. TFINDS makes the data in this item accessible to the next instructions in sequence in the program.

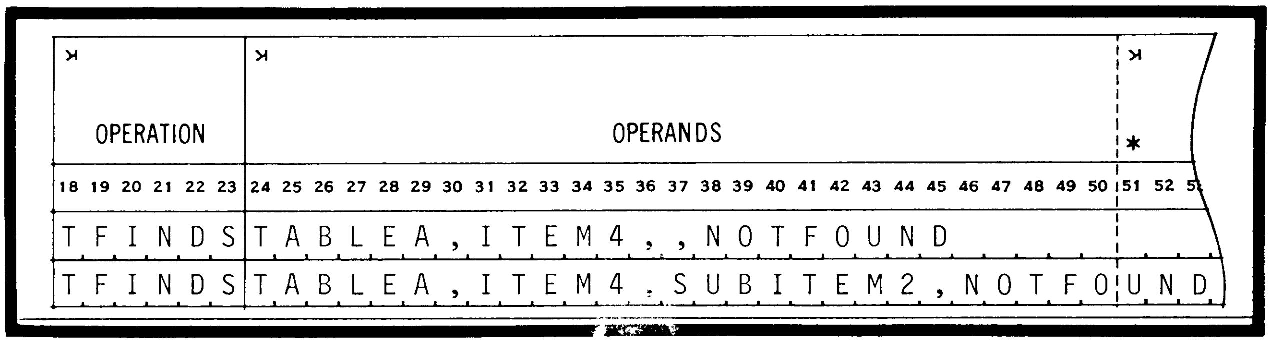
If, however, the key of the item selected in the table is greater than the key of the desired item, TFINDS makes this item accessible, sets the equal (E) flag, and transfers control to the user's routine specified by the branch operand.

If the items in the table are exhausted before an equal-to or greater-than condition is met, TFINDS makes accessible the position immediately following the last item in the table, sets the greater (G) flag, and transfers control to the user's routine specified by the branch operand.

If the slot processing technique is used on this table, TFINDS ignores each nonactive item it encounters and selects the next item in sequence.

When the keys are in descending sequence, the greater-than and less-than rules are reversed.

### Example



| OPERATION | OPERANDS | * |
|---|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 | 51 52 5 |
| TFINDS | TABLEA,ITEM4,,NOTFOUND | |
| TFINDS | TABLEA,ITEM4,SUBITEM2,NOTFO | UND |

In positions 18 - 23, enter TFINDS.

As the first operand, enter the reference of the table to be accessed (TABLEA in this example).

As the second operand, enter the reference of the field containing the data (ITEM4 in this example) to be compared with the major key of each item in the table.

As the third (optional) operand, enter the reference of the field containing the data (SUBITEM2 in this example) to be compared with the minor key of each item in the table. If a 1-key format is used, enter a comma to indicate that no minor key comparison is made.

As the branch operand, enter the reference of the user's routine to receive control (such as NOTFOUND) when the desired item is beyond the range of the table or is within the range but physically missing from the table.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

In the following illustrations, the items are arranged in ascending sequence of keys. TFINDS is to search the table for the item with a key of 4. If the item is not in the table, TFINDS is to transfer control to the routine referenced by NOTFOUND. An asterisk indicates the first item that TFINDS selects.

| TABLE A | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 1 | 1 is less than 4. | Select next item. |
| Key / Data | 2 | 2 is less than 4. | Select next item. |
| Key / Data | 3 | 3 is less than 4. | Select next item. |
| Key / Data | 4 | 4 is equal to 4. | Make this item accessible. |

| TABLE B | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 1 | 1 is less than 4. | Select next item. |
| Key / Data | nonactive | | Select next item. |
| Key / Data | 3 | 3 is less than 4. | Select next item. |
| Key / Data | 5 | 5 is greater than 4. | Make this item accessible, set E flag, branch to NOTFOUND. |

In the following illustration, the keys are arranged in descending sequence of keys. TFINDS is to make accessible the item with a key of 4. If the item is not in the table, TFINDS is to transfer control to the routine referenced by NOTFOUND. An asterisk indicates the first item that TFINDS selects.

| TABLE C | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 8 | 8 is greater than 4. | Select next item. |
| Key / Data | 7 | 7 is greater than 4. | Select next item. |
| Key / Data | nonactive | | Select next item. |
| Key / Data | 5 | 5 is greater than 4. | Select next item. |
| *end of table | | | Set G flag, branch to NOTFOUND. |

If the NOTFOUND routine executes a TSERT instruction, the new item with a key of 4 is inserted immediately after the item with a key of 5, thereby extending the length of the table.

In the following illustration, the keys are arranged in descending sequence of keys. TFINDS is to make accessible the item with a key of 4. If the item is not in the table, TFINDS is to transfer control to the routine referenced by NOTFOUND. An asterisk indicates the first item that TFINDS selects.

| TABLE D | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 6 | 6 is greater than 4. | Select next item. |
| Key / Data | 5 | 5 is greater than 4. | Select next item. |
| Key / Data | 3 | 3 is less than 4. | Make this item accessible, set E flag, branch to NOTFOUND. |

## Conventions

The items in the table must be arranged in either ascending or descending sequence of keys.

When the TFINDS instruction is used to locate the position in the table of where an item is to be inserted, control is transferred to the branch operand in two instances:

- If the key of the item to be inserted is within the range of the keys currently in the table, TFINDS sets the E flag and transfers control to the branch routine.
- If the key of the item to be inserted exceeds the key of the last item in the table, TFINDS sets the G flag and transfers control to the branch operand.

If the user wishes to process each of these conditions differently, he may code a BRG or a BRE as the first instruction in the branch routine, thereby setting up a routine for an equal condition and a routine for a greater-than condition. However, in any instance, the execution of a TSERT inserts the item into its proper location.

The end results of both TFINDS and TFINDB are the same; however, processing time differs. If a table has a large number of sequentially organized items, a binary search (TFINDB) proves quickest in accessing the individual items. If the table has a small number of sequentially organized items, a sequential search (TFINDS) is optimal.

TFINDS may be executed on all types of tables.

Literals are allowed in both the second and third operands of TFINDS. If used, their length will be that of the image shown, i.e. a literal of XYZ must be expressed as 'XYZ'. X is the assumed type.

References used in the second and third operands of this instruction must not be contained within the table referenced by the first operand.

## TFINDB

### Function

TFINDB assumes that TBEGF has previously been executed. The items in the table must be arranged in either ascending or descending sequence of keys.

TFINDB performs a binary search for the desired item. It begins its search with an item in the last half of the table (the position of this item is calculated by TFINDB) and compares the keys of this item with the keys of the desired item (the contents of the second and optionally third operands).

TFINDB determines whether the desired item is above or below the item selected in the table and selects, in the appropriate direction, an item half the distance between the uncompared items. It then compares the keys of this item with the keys of the desired item.

If the slot processing technique is used on this table, TFINDB ignores each nonactive item it encounters and selects in the appropriate direction the next item in sequence.

TFINDB continues to select the item half the distance between the uncompared items in the table until a hit occurs. TFINDB then makes this item accessible to the program.

If the desired item is within the range of the items in the table but is not physically in the table, TFINDB sets the equal (E) flag, and makes accessible the next higher item (if the keys are in ascending sequence) or the next lower item (if the keys are in descending sequence). TFINDB then transfers control to the user's routine specified by the branch operand.

If the desired item is not within the range of the items in the table, TFINDB sets the greater (G) flag and transfers control to the user's routine specified by the branch operand.

### Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TFINDB | TABLEC,ITEM2,,NOTFOUND |
| TFINDB | TABLEC,ITEM2,SUB1,NOTFOUND |

In positions 18 - 23, enter TFINDB.

As the first operand, enter the reference of the table to be accessed (TABLEC in this example).

As the second operand, enter the reference of the field containing the data (ITEM2 in this example) to be compared with the major key of each item in the table.

As the third (optional) operand, enter the reference of the field containing the data (SUB1 in this example) to be compared with the minor key of each item in the table. If a 1-key format is used, enter a comma to indicate that no minor key comparison is made.

As the branch operand, enter the reference of the user's routine to receive control (such as NOTFOUND) when the desired item either is beyond the range of the table or is within the range but physically missing from the table.

NOTE

> References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

In the following illustration, TFINDB is to make accessible the item with the key of 15. TFINDB begins its search by selecting the eighth item in sequence.

| ITEM NO. | TABLE A | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|---|
| 1 | Key / Data | 2 | | |
| 2 | Key / Data | 3 | | |
| 3 | Key / Data | 6 | | |
| 4 | Key / Data | 8 | 8 is less than 15. | Go down, select item 6. |
| 5 | Key / Data | 10 | | |
| 6 | Key / Data | 13 | 13 is less than 15. | Go down, select item 7. |
| 7 | Key / Data | 15 | 15 is equal to 15. | Set flag, make this item accessible. |
| *8 | Key / Data | 17 | 17 is greater than 15. | Go up, select item 4. |
| 9 | Key / Data | 21 | | |

In the following illustration, the table is arranged in ascending sequence of keys. TFINDB is to make accessible the item with a key of 22. If the item is not in the table, TFINDB is to transfer control to a routine referenced by NOTFOUND. TFINDB begins its search by selecting the eighth item in sequence.

| ITEM NO. | TABLE C | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|---|
| 1 | Key / Data | 2 | | |
| 2 | Key / Data | 3 | | |
| 3 | Key / Data | 6 | | |
| 4 | Key / Data | 8 | | |
| 5 | Key / Data | 10 | | |
| 6 | Key / Data | 13 | | |
| 7 | Key / Data | 15 | | |
| *8 | Key / Data | 17 | 17 is less than 22. | Go down, select item 9. |
| 9 | Key / Data | 21 | 21 is less than 22. | Set G flag. branch to NOTFOUND. |
| *end of table | | | | |

In the following illustration the items are arranged in ascending sequence of keys. TFINDB is to make accessible the item with the key of 15. TFINDB begins its search by selecting the eighth item in sequence.

| ITEM NO. | TABLE B | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|---|
| 1 | Key / Data | 2 | | |
| 2 | Key / Data | 3 | | |
| 3 | Key / Data | 6 | 6 is less than 15. | Go down, select item 6. |
| 4 | Key / Data | nonactive | | Keep going up, select next item in sequence (3) |
| 5 | Key / Data | 10 | | |
| 6 | Key / Data | nonactive | | Keep going down, select next item in sequence (7) |
| 7 | Key / Data | 15 | 15 is equal to 15. | Set E flag, make this item accessible. |
| *8 | Key / Data | 17 | 17 is greater than 15. | Go up, select item 4. |
| 9 | Key / Data | 21 | | |

Conventions

The items in the table must be arranged in either ascending or descending sequence of keys.

The end results of both TFINDS and TFINDB are the same; however, processing time differs. If a table has a large number of sequentially organized items, a binary search (TFINDB) proves quickest in accessing the individual items. If the table has a small number of sequentially organized items, a sequential search (TFINDS) is optimal.

TFINDB may be executed on all types of tables.

Literals are allowed in both the second and third operands of TFINDB. If used, their length will be that of the image shown, i.e. a literal of 12A must be expressed as '12A'. X is the assumed type.

References used in the second and third operands of this instruction must not be contained within the table referenced by the first operand.

## TFINDP

### Function

TFINDP assumes that TBEGF has previously been executed.

TFINDP selects the previous sequential item in the table. The first item selected by TFINDP is one position prior to the previously executed table instruction. If the table uses slot processing, TFINDP selects the previous active item. Data in the item selected by TFINDP can be accessed by the program.

If the currently accessible item is the first item in the table, TFINDP cannot select another item. When this situation occurs, TFINDP makes the first item in the table accessible to the program and transfers control to the routine specified by the branch operand.

The TFINDP instruction is useful when the programmer desires to select a previous item in the table without beginning a search at the first item. TFINDP makes an item accessible to the next instruction in sequence in the program.

### Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TFINDP | TABLED,TOOFAR |

In positions 18 - 23, enter TFINDP.

As the first operand, enter the reference of the table to be accessed, TABLED in this example.

As the branch operand, enter the reference of the user's routine to receive control, such as TOOFAR, when the item accessed is the first item in the table.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

### Conventions

TFINDP can be executed on all types of tables.

Literal operands are not permitted in TFINDP.

## TFINDO

### Function

TFINDO assumes that TBEGF has previously been executed. The items in the table must be arranged in either ascending or descending sequence of keys.

TFINDO performs a sequential search for the desired item by key comparison. It begins searching at the position determined by the previously executed table instruction; i.e., it begins where the previous search ended. TFINDO compares the key of the item selected with the key of the desired item (the contents of the second operand). If the compared fields are equal, TFINDO makes the data in this item accessible to the next instructions in sequence in the program.

If the keys are in ascending sequence, TFINDO continues to search until it reaches a key greater than the key of the desired item. TFINDO makes this table item accessible, sets the equal (E) flag, and transfers control to the user's routine specified by the branch operand. If the items in the table are exhausted before an equal-to or greater-than condition is met, TFINDO makes accessible the position immediately following the last item in the table, sets the greater (G) flag, and transfers control to the user's routine specified by the branch operand.

NOTE

When the keys are in descending sequence, the greater-than and less-than rules are reversed.

If the table uses slot processing, TFINDO ignores each nonactive item it encounters and selects the next item in sequence.

### Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TFINDO | TABLEA,ITEM8,,NOTFOUND |
| TFINDO | TABLEA,ITEM8,SUB3,NOTFOUND |

In positions 18 - 23, enter TFINDO.

As the first operand, enter the reference of the table to be accessed (TABLEA in this example).

As the second operand, enter the reference of the field containing the data (ITEM8 in this example) to be compared with the major key of each item in the table.

As the third (optional) operand, enter the reference of the field containing the data (SUB3 in this example) to be compared with the minor key of each item in the table. If a 1-key format is used, enter a comma to indicate that no minor key comparison is made.

As the branch operand, enter the reference of the user's routine to receive control (such as NOTFOUND) when the desired item either is beyond the range of the table or is within the range but physically missing from the table.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

In the following illustrations, the items are arranged in ascending sequence of keys. TFINDO is to search for the item with the key of 8. If the item is not in the table, TFINDO is to transfer control to the routine referenced by NOTFOUND. An asterisk indicates the first item that TFINDO selects (the item determined by the previously executed table instruction).

| TABLE A | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 6 | 6 is less than 8. | Select next item. |
| Key / Data | 7 | 7 is less than 8. | Select next item. |
| Key / Data | 8 | 8 is equal to 8. | Make this item accessible. |

| TABLE B | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 5 | 5 is less than 8. | Select the next item. |
| Key / Data | nonactive | | Select the next item. |
| Key / Data | 7 | 7 is less than 8. | Select the next item. |
| Key / Data | 9 | 9 is greater than 8. | Make this item accessible, set the E flag, branch to NOTFOUND. |

In the following illustration, the keys are arranged in descending sequence of keys. TFINDO is to search this table for the item with a key of 8. If the item is not found, TFINDO is to transfer control to the routine referenced by NOTFOUND. An asterisk indicates the first item that TFINDO selects.

| TABLE C | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 12 | 12 is greater than 8. | Select next item. |
| Key / Data | 11 | 11 is greater than 8. | Select next item. |
| Key / Data | nonactive | | Select next item. |
| Key / Data | 9 | 9 is greater than 8. | Select next item. |
| End of table | | | Set G flag, Branch to NOTFOUND. |

NOTE

If the NOTFOUND routine executes a TSERT instruction, the new item with a key of 8 is inserted immediately after the item with a key of 9, thus extending the length of the table.

In the following illustration, the keys are arranged in descending sequence of keys. TFINDO is to make accessible the item with a key of 8. If the item is not in the table, TFINDO is to transfer control to the routine referenced by NOTFOUND. An asterisk indicates the first item that TFINDO selects.

| TABLE D | CONTENTS OF KEY | COMPARISON | ACTION |
|---|---|---|---|
| *Key / Data | 10 | 10 is greater than 8. | Select next item. |
| Key / Data | 9 | 9 is greater than 8. | Select next item. |
| Key / Data | 7 | 7 is less than 8. | Make this item accessible, set E flag, branch to NOTFOUND. |

## Conventions

The items in the table must be arranged in either ascending or descending sequence of keys.

When the TFINDO instruction is used to locate the position in the table where an item is to be inserted, control is transferred to the branch operand.

- If the key of the item to be inserted is within the range of the keys currently in the table, TFINDO sets the E flag and transfers control to the branch routine.
- If the key of the item to be inserted exceeds the key of the last item in the table, TFINDO sets the G flag and transfers control to the branch routine.

If the user wishes to process each of these conditions differently, he may code a BRG or BRE as the first instruction in the branch routine. This procedure sets up a routine for an equal condition and a routine for a greater-than condition. The execution of a TSERT instruction inserts the item into its proper location.

The end results of TFINDO, TFINDS, and TFINDB are the same; however, processing time differs. If a table has a large number of sequentially organized items, a binary search (TFINDB) is the fastest method of accessing the individual items. Either TFINDO or TFINDS is preferable when the table has a small number of sequentially organized items. TFINDO is used to begin the search without returning to the first item in the table, thus reducing processing time.

TFINDO may be executed on all types of tables.

Literals are allowed in both the second and third operands of TFINDO. If used, their length will be that of the image shown; i.e., a literal of XYZ must be expressed as 'XYZ'. X is the assumed type.

References used in the second and third operands of this instruction must not be contained within the table referenced by the first operand.

## TDEL INSTRUCTION

### TDEL

### Function

The TDEL (delete) instruction assumes that a variation of the TFIND instruction has first located the desired item that TDEL is to delete. Then TDEL deletes the item from the table. The item immediately following the deleted item is now accessible to the program.

If, however, the source line contains the optional second operand, TDEL first moves the contents of the current item into the field specified by this operand and then deletes the current item from the table.

If the table uses the slot processing technique, TDEL deletes the item by making it nonactive.

If the table uses the pushdown processing technique, TDEL contracts the length of the table and adjusts the table length indicator (and the variable length indicator if the table is within a record).

If no items remain in the table or if the item to be deleted is already non-active, TDEL transfers control to the routine specified by the branch operand.

### Example 1

If TDEL is to move the information in the item to be deleted into another field before deleting the item from the table, code the instruction in the following format.

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TDEL | TABLEA,ITEM6,EXCEPT |

In positions 18 - 21, enter TDEL.

As the first operand, enter the reference of the table (TABLEA in this example) which contains the item to be deleted.

As the second operand, enter the reference of the field (ITEM6 in this example) which is to receive the deleted item.

As the branch operand, enter the reference of the user's routine to receive control (EXCEPT in this example) if no items remain in the table or if the item to be deleted is already nonactive.

Example 2

If TDEL is only to delete the item from the table, code the instruction in the following format.

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TDEL | TABLEB,,EXCEPT |

In positions 18 - 21, enter TDEL.

As the first operand, enter the reference of the table (TABLEB in this example) which contains the item to be deleted.

As the branch operand, enter the reference of the user's routine to receive control (EXCEPT in this example) if no items remain in the table or if the item to be deleted is already nonactive.

NOTE

> If this format is used, two commas must separate the table reference from the branch operand. References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

Conventions

TDEL can be executed on all types of tables.

Before a TDEL can be executed, the item to be deleted from the table must be made accessible.

NOTE

> Literal operands are not permitted in TDEL.

## TSERT INSTRUCTION

### TSERT

#### Function

TSERT can only be executed on a variable-length table.

The TSERT instruction assumes that a TFIND instruction has first located the desired position into which TSERT is to insert a new item. Before it inserts this item, TSERT uses either the pushdown or the slot processing technique (determined by the structure of the specified table) to make room for the new item. It then inserts into this vacant position the new item specified by the second operand.

If the table is a freestanding, variable-length table that uses the slot technique (option 4 in position 57 of the table specification sheet), TSERT first checks the item immediately preceding the item made available by TFIND. If this item is nonactive, TSERT inserts the new item into this location. If it is active, TSERT follows the normal rules. (It uses the slot technique to vacate the location made available by TFIND and then inserts the new item into this vacant position.)

After TSERT inserts the item into its proper position, it updates the contents of the table length indicator. If the table is within a record, TSERT also alters the contents of the variable length indicator.

If the specified table has the optional item counter, TSERT increments it by one each time it inserts a new item into the table.

The newly inserted item is now accessible to the program.

If, however, the length of the table is currently the maximum length allowed or if the insertion of this new item will extend the length of the table beyond the maximum length allowed, TSERT transfers control to the routine specified by the branch operand. The item made accessible by TFIND is still accessible.

Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TSERT | TABLED,DATAFIELD,TOOFAR |

In positions 18 - 22, enter TSERT.

As the first operand, enter the reference of the table (TABLED in this example) into which the item is to be inserted.

As the second operand, enter the reference of the field (DATAFIELD in this example) containing the item to be inserted into the table.

As the branch operand, enter the reference of the routine to receive control (TOOFAR in this example) if the current table length is the maximum length allowed.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

Conventions

The table must be variable in length.

A TFIND instruction must first locate the desired position in the table into which TSERT is to insert a new item.

TSERT may be executed on the following types of tables:

- Variable-length table within an area using the slot processing technique (structure 4).
- Variable-length table within an area using the pushdown processing technique (structure 5).
- Variable-length table within a record using the pushdown processing technique (structure 6).

Literals are allowed in the second operand of TSERT. If used, their length will be that of the image shown, i.e. a literal of A10 (length of 3) must be expressed as 'A10'. X is the assumed type.

## TPACK INSTRUCTION

### TPACK

#### Function

TPACK can be executed only on tables using the slot processing technique. Its main purpose is to decrease the access time needed to search a table.

TPACK moves all active items to the beginning of the table without disturbing the sequence in which these items are currently ordered. Thus, the instruction groups all active items at the beginning of the table and groups all nonactive items at the end of the table.

If the table contains no active items, TPACK transfers control to the routine specified by the branch operand.

The items in TABLEA are arranged in the following sequence of keys (na signifies a nonactive item).

**TABLEA BEFORE TPACK**

| | | | | | | |
|---|---|---|---|---|---|---|
| na | na | 4 | 5 | na | 25 | na |

After TPACK has been executed on TABLEA, the items in TABLEA are arranged in the following sequence of keys.

**TABLEA AFTER TPACK**

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 5 | 25 | na | na | na | na |

Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TPACK | TABLEA,NONACTIVE |

In positions 18 - 22, enter TPACK.

As the first operand, enter the reference of the table to be accessed (TABLEA in this example).

As the branch operand, enter the reference of the routine to receive control (NONACTIVE in this example) if the specified table contains no active items.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

Conventions

TPACK may be executed on the following tables:

- Fixed-length table within an area using the slot processing technique (structure 1).
- Fixed-length table within a record using the slot processing technique (structure 2).
- Fixed-length minor table using the slot processing technique (structure 3).
- Variable-length table within an area using the slot processing technique (structure 4).

NOTE

Literal operands are not used with TPACK.

## TSORT INSTRUCTIONS

There are two variations of the TSORT instruction, TSORTA and TSORTD. Each is discussed separately.

### TSORTA

#### Function

TSORTA sorts the items in the specified table into ascending sequence of keys.

The items in TABLEA are arranged in the following sequence of keys.

**TABLEA BEFORE TSORTA**

| | | | | | | |
|---|---|---|---|---|---|---|
| 23 | 7 | 32 | 19 | 20 | 6 | 2 |

After TSORTA has been executed on TABLEA, the items in the table are arranged in the following sequence of keys.

**TABLEA AFTER TSORTA**

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 6 | 7 | 19 | 20 | 23 | 32 |

#### Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| T S O R T A | T A B L E A |

## Conventions

TSORTA can be executed on all types of tables. If the table is within a variable-length record, the record must be moved to a workarea before TSORTA can be executed.

TSORTA requires that a workarea of one item length follow the items in the table. For instance, if a table is to contain a maximum of ten 10-character items and if TSORTA is to be executed on the table, allow within the table an additional item length (ten characters in this instance) to be used as the workarea. Do this by the following entries:

1. Specify on the table specification sheet that the maximum length of the table is 100 characters.
2. Specify on the data layout sheet that the item-length is 10 characters.
3. Allow 110 characters within the record or the area for the table.

NOTE

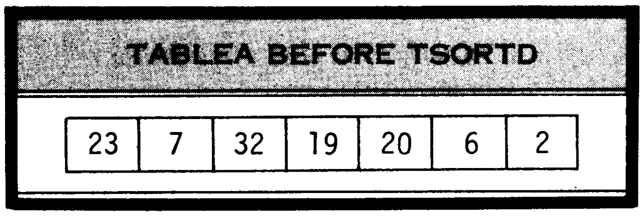
Literal operands are not permitted in TSORTA.

## TSORTD

### Function

TSORTD sorts the items in the specified table into descending sequence of keys.

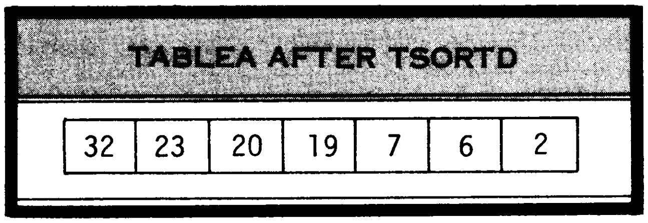
The items in TABLEA are arranged in the following sequence of keys.



**TABLEA BEFORE TSORTD**

| | | | | | | |
|---|---|---|---|---|---|---|
| 23 | 7 | 32 | 19 | 20 | 6 | 2 |

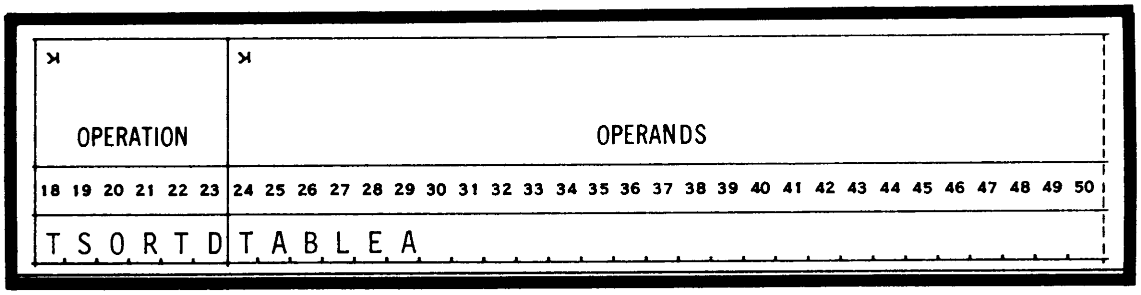
After TSORTD has been executed, the items in TABLEA are arranged in the following sequence of keys.



**TABLEA AFTER TSORTD**

| | | | | | | |
|---|---|---|---|---|---|---|
| 32 | 23 | 20 | 19 | 7 | 6 | 2 |

### Example



| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TSORTD | TABLEA |

## Conventions

TSORTD can be executed on all types of tables. If the table is within a variable-length record, the record must be moved to a workarea before TSORTD can be executed.

TSORTD requires that a workarea of one item length follow the items in the table. For instance, if a table is to contain a maximum of ten 10-character items and if TSORTD is to be executed on the table, allow within the table an additional item length (10 characters in this instance) to be used as the workarea. Do this by the following entries:

1. Specify on the table specification sheet that the maximum length of the table is 100 characters.
2. Specify on the data layout sheet that the item-length is 10 characters.
3. Allow 110 characters within the record or the area for the table.

NOTE

Literal operands are not permitted in TSORTD.

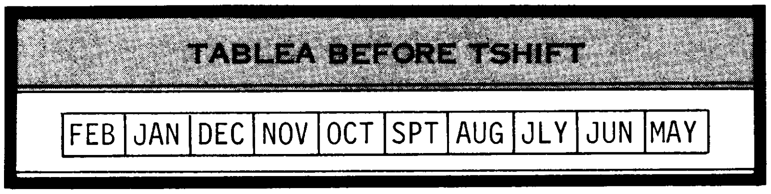
## TSHIFT INSTRUCTION

### TSHIFT

### Function

The TSHIFT instruction shifts the items in the table one item length toward the end of the table and inserts the item specified by the second operand into the first position of the table. The last item in the table is destroyed.

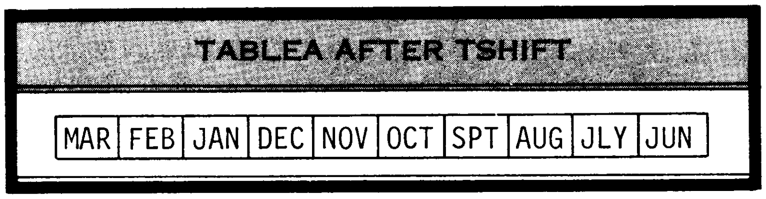
The items in TABLEA are arranged in the following sequence of keys.



**TABLEA BEFORE TSHIFT**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| FEB | JAN | DEC | NOV | OCT | SPT | AUG | JLY | JUN | MAY |

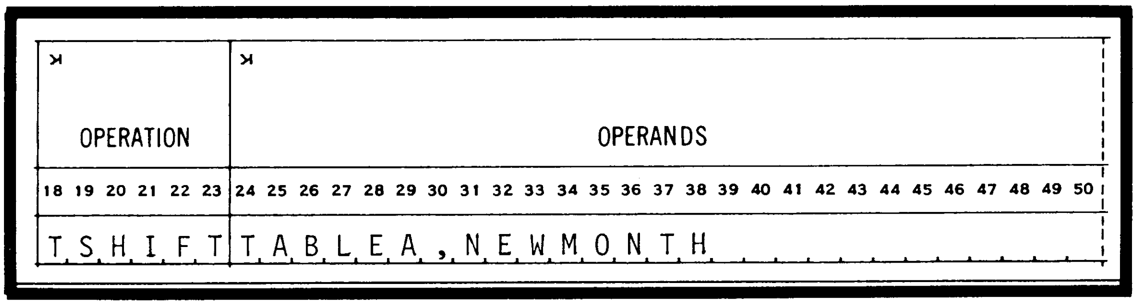
Assume that the contents of NEWMONTH is MAR. After a TSHIFT TABLEA, NEWMONTH instruction is executed, the items in TABLEA are arranged in the following sequence of keys.



**TABLEA AFTER TSHIFT**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| MAR | FEB | JAN | DEC | NOV | OCT | SPT | AUG | JLY | JUN |

### Example



| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TSHIFT | TABLEA,NEWMONTH |

In positions 18 - 22, enter TSHIFT.

As the first operand, enter the reference of the table to be accessed (TABLEA in this example).

As the second operand, enter the reference of the field (NEWMONTH in this example) containing the item to be inserted into the first position of the table.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

Conventions

- Literals are allowed in the second operand of TSHIFT. If used, their length will be that of the image shown, i.e. a literal of AAA (length of 3) must be expressed as 'AAA'. X is the assumed type.
- TSHIFT can be executed on all types of tables.

## TMARK INSTRUCTION

TMARK

### Function

The TMARK instruction takes the relative memory address of the item currently accessible and stores it in the memory area specified by the second operand.

TRESET is the logical complement of TMARK.

### Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TMARK | TABLEC,DATAFIELD |

In positions 18 - 22, enter TMARK.

As the first operand, enter the reference of the table to be accessed (TABLEC in this example).

As the second operand, enter the reference of a 3-character binary field (DATAFIELD in this example) in a memory area into which TMARK stores the memory address of the item currently accessible.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

### Conventions

TMARK can be executed on all types of tables.

Literal operands are not permitted in TMARK.

## TRESET INSTRUCTION

TRESET

Function

TRESET makes accessible the item located at the address that TMARK has stored in the field referenced by the second operand.

However, if this location is currently not within the boundaries of the table, TRESET transfers control to the user's routine specified by the branch operand.

TMARK is the logical complement of TRESET.

Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TRESET | TABLEB,DATAFIELD,NOTFOUND |

In positions 18 - 23, enter TRESET.

As the first operand, enter the reference of the table to be accessed (TABLEB in this example).

As the second operand, enter the reference of a 3-character binary field (DATAFIELD in this example) in a memory area into which TMARK has stored the address of an item in the table.

As the branch operand, enter the reference of the routine to receive control (NOTFOUND in this example) if the address in the field referenced by the second operand is no longer within the table boundaries.

NOTE

References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

## Conventions

Literals are allowed in the second operand of TRESET. If used, their length will be that of the image shown, i.e. a literal of 001 (length of 3) must be expressed as '001'. B (binary) is the assumed type.

TRESET can be executed on all types of tables.

## TJUMP INSTRUCTION

### TJUMP

### Function

The TJUMP instruction provides the capability for the program to transfer control to a specific routine based on the program's current position in a table. To do this TJUMP makes two assumptions: first, that a preceding TFIND instruction has located a desired item in a table; second, that a list of transfer-of-control instructions exists.

TJUMP references the table and determines the relative location of the item position by TFIND, for example, the fourth item in the table.

TJUMP then references the list of transfer-of-control instructions and locates the instruction at the same relative location as the item positioned in the table, i.e. the fourth instruction in the list.

There must be a 4-character, transfer-of-control instruction for each item to be accessed in the table, i.e. a 100 item table requires that 100 transfer-of-control instructions be listed. Each transfer-of-control instruction must be either a LINK or an unconditional BR. If the transfer-of-control instruction is not a LINK or BR, the E flag is set and control is transferred to the user's routine specified in the branch operand (of the TJUMP instruction).

If the items in the table are exhausted before an equal-to or greater-than condition is met, the G flag is set to indicate this off-table condition, and control is transferred to the user's routine specified by the branch operand (of the TJUMP instruction).

If all conditions are met, the jump from table to list is made and control is passed to the routine specified by the appropriate transfer-of-control instruction.

### Example

| OPERATION | OPERANDS |
|---|---|
| 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 |
| TJUMP | TRANSCODE,ITEM2,ROUTINEC |

In positions 18 - 22, enter TJUMP.

As the first operand, enter the reference of the table to be accessed (TRANSCODE in this example).

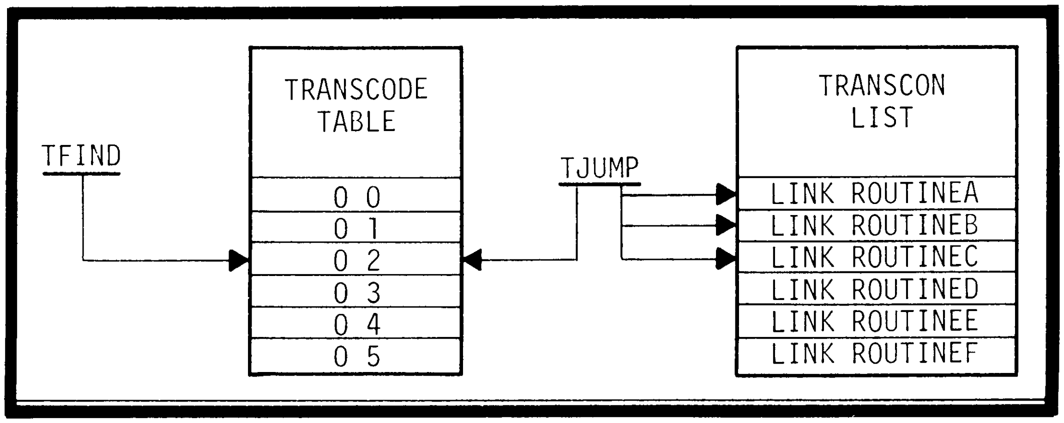
As the second operand, enter the reference of the first 4-character instruction in the transfer-of-control list (ITEM2 in this example). Control would be returned to this instruction if the table reference in the first operand was positioned on the first item.

As the branch operand, enter the reference of the user's routine to receive control (ROUTINEC in this example) if the transfer-of-control instruction is other than a LINK or BR or if the table reference in the first operand finds an off-table condition.

NOTE

> References in the operands column may be as long as 10 characters and may extend into the comments column when necessary.

For example, assume the following table of transaction codes and list of transfer-of-control instructions exist.



- A TFIND instruction has made the third item, 02, in the TRANSCODE table accessible.
- TJUMP references the TRANSCODE table and determines the third item is accessible.
- TJUMP references the TRANSCON list and positions itself at the first instruction and correlates to the third instruction.
- A LINK is made and control is passed to ROUTINEC.

## Conventions

A list of transfer-of-control instructions must exist and contain an instruction for each item accessed in the table.

Literal operands are not permitted in TJUMP.

TJUMP can be executed on all types of tables.