CURSO: LENGUAJE COBOL ENFOCADO A LA MAQUINA VAX -11
DEL 2 AL 6 DE DICIEMBRE DE 1985
DIRIGIDO AL PERSONAL PROFESIONAL DE DIRECCION. GRAL. DE DESAROLLO TECNOLOGICO S.C.T
MEXICO. D.F

COBOL BASICO

DICIEMBRE DE 1985.

Apuntes de

COBOL BASICO

COBOL

Common Business Oriented Lansuage

INTRODUCCION

El COBOL es un lensuaje que sirve para programar computadoras digitales, orientado al procesamiento de información para aplicaciones comerciales, diseñado especialmente para servir en la administración.

El lensuaje COBOL tiene las siguientes características:

Es un lensuaje de Alto Nivel: emplea términos de la lensua inslesa, lo cual facilita la programación.

Es de Carácter Universal: la masoría de las computadoras lo aceptan y es similar en todas ellas.

Es Autodocumentado: la extructura de los programas obligan a la documentación de los mismos.

Pertenece a los Lensuajes Compiladores: se requiere de un programa (compilador) que traduzca del lensuaje COBOL al lensuaje de máquina de la computadora.

CARACTERES

Los elementos básicos de cualquier lensuaje son los caracteres con los cuales éste se forma.

En el lensuaje COBOL los caracteres que se emplean son:

Caracteres Númericos : 0 a 9

Caracteres Alfabéticos: A a la Z

Ñ (espacio en blanco)

Caracteres Especiales : +-/=$,;.:'()

En algunos casos, en el lenguaje COBOL, ciertos caracteres tiene un sisnificado especial, estos caracteres son:

CARACTERES DE EDICION:

B Inserción de espacios o blancos
Z Supresión de ceros
0 Inserción de ceros
+ Signo Positivo
- Signo negativo
CR Crédito
DB Débito
* Protección de cheques
$ Signo de pesos
, Inserción de comas
. Inserción de puntos

CARACTERES DE PUNTUACION:

, Coma
; Punto y coma
: Dos Puntos
. Punto
" Comillas
(Paréntesis izquierdo
) paréntesis derecho
Ø Espacio/o blanco
/ diagonal

OPERADORES ARITMETICOS

- Menos
+ Más
* Multiplicación
/ División
** Exponenciación

CARACTERES DE RELACION

> Mayor
< Menor
= Igual

PROGRAMACION

En la elaboración de cualquier programa en COBOL es necesario definir lo siguiente:

a) Los Resultado a obtenerse (Salidas).

b) Los Datos requeridos para obtener los resultados deseados (Entradas).

c) La forma en que se procesarán los datos (Algoritmos).

d) Los dispositivos externos al programa de los cuales se obtendrán los datos a procesarse (Dispositivos de entrada).

e) Los dispositivos externos al programa en los cuales se emitiran los resultados (Dispositivos de Salida).

f) Los dispositivos de almacenaminto de información, etcétera.

En la definición de la información que se procesará hay que especificar las características de ésta, como es el tipo (alfabético, numérico, etc.), y el tamaño (cuantos caracteres máximo puede tener un dato, por ejemplo un nombre).

En todo programa existe cierta información cuyo contenido es definido dentro del programa (tales como los encabezados de los reportes que se emitirán) y otra información cuyo contenido (mas no sus características) es proporcionado al momento de ejecución del programa (por ejemplo, los nombres de los empleados en la elaboración de una nomina). En este último caso los programas son practicamente independientes del contenido de la información a procesar.

ESTRUCTURA DE UN PROGRAMA

Todo programa en COBOL está estructurado en cuatro divisiones en donde se definen lo siguiente:

1) La identificación del programa

2) La asociación al programa de los dispositivos externos por los cuales se obtendrán los datos de entrada y se emitirán los resultados.

3) La descripción de la información que se procesará.

4) Los algoritmos de procesamiento de la información.

En el lenguaje COBOL, a diferencia de otros lenguajes, es necesario entender la función de cada una de las Divisiones antes de poder desarrollar algún programa.

DESCRIPCION DE LA INFORMACION

Para definir en un programa en COBOL, la información que se va a procesar, tanto de entrada, de salida, como intermedia, se emplean los conceptos Campos de Datos, los Registros y los Archivos.

CAMPOS DE DATOS

Los Campos de Datos definen áreas de almacenamiento de información en el programa. A cada campo de dato hay que definirle el número de caracteres que puede contener (longitud del campo) y el tipo de información que podrá almacenar (sea alfabético, numérico o alfanumérico).

Por ejemplo el campo para almacenar el número de cuenta de un alumno de la UNAM, tendrá una longitud de 8 caracteres del tipo numérico.

NOMBRE-DE-DATO

Es el nombre asociado a un Campo de información, para hacerle referencia dentro de un programa en COBOL (ejemplo: NO-DE-CUENTA)

REGLAS PARA FORMAR LOS NOMBRES DE DATO

Las reglas para formar los datos de dato y en general para formar cualquier nombre que el programador tenga que definir dentro de un programa son:

a) De 1 a 30 caracteres elegidos de los siguientes:

Á a la Z (excepto Ñ) 0 al 9 - (guiones)

b) Deben empezar con un carácter alfabético (letra)

c) Los guiones (-) sólo podrán ir intermedios en el nombre.

No pueden ir al principio ni al final del nombre.

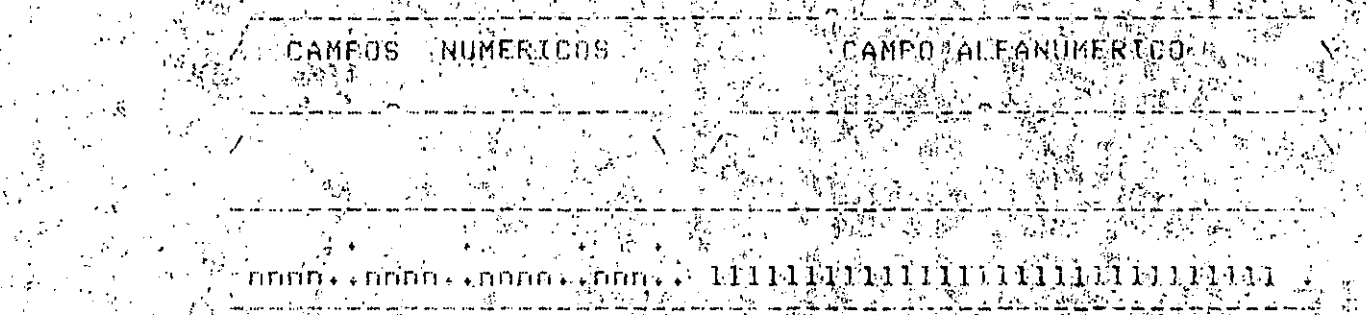
d) No se permiten espacios o blancos intermedios.

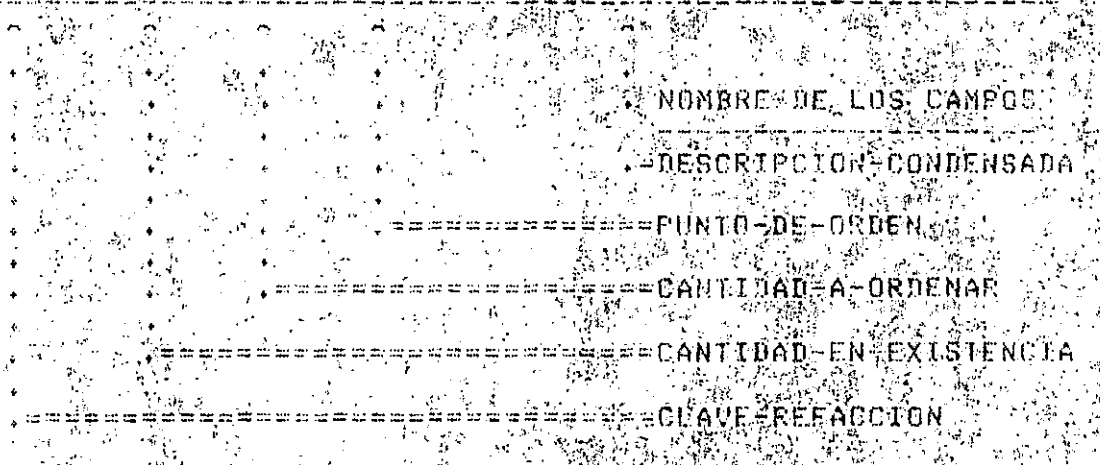
e) No deben emplearse palabras reservadas (ver más adelante) en la elaboración de los nombres.

REGISTRO

Un registro es un conjunto de campos de información que se agrupan por su relación, y se referencian en conjunto con un nombre (nombre de registro).

Ejemplo de un registro de inventario identificado con el nombre de REGISTRO-DE-INVENTARIO:





Este ejemplo se codifica en lenguaje COBOL como sigue:

```
01 REGISTRO-DE-INVENTARIO.
   03 CLAVE-REFACCION          PIC 9 (4).
   03 CANTIDAD-EN-EXISTENCIA   PIC 9 (4).
   03 CANTIDAD-A-ORDENAR       PIC 9 (4).
   03 PUNTO-DE-REORDEN         PIC 9 (3).
   03 DESCRIPCION-CONDENSADA   PIC X (30).
```

ARCHIVO

Un Archivo es un conjunto de registros que tienen el mismo tipo de información. A los archivos se les asigna un Nombre para hacerles referencia dentro del Programa (Nombre de Archivo).

Los Archivos siempre se relacionan con dispositivos externos al programa, conocidos como equipos periféricos (ejemplo: Lectora de Tarjetas, Impresoras, Cintas y Discos magnéticos, etc.).

A través de los Archivos se hace la Entrada (obtención de los datos a proceso) y la Salida (emisión de reportes y resultados) de la información de cualquier programa en COBOL.

ALGORITMO DE PROCESO.

El algoritmo de proceso de la información la componen una serie de instrucciones denominadas oraciones, que se organizan en forma lógica para obtener los resultados deseados.

VERBOS.

Las oraciones están compuestas con Verbos, que como en todo lenguaje ordenan una acción. El COBOL proporciona una serie de Verbos, cuya sintaxis y función se encuentran ya definidos en el lenguaje (pertenecen a las palabras reservadas, ver mas adelante).

Algunos verbos operan con campos de información y otros definen el órden en que se ejecutarán las instrucciones. Con los Verbos se pueden hacer:

a) Operaciones Aritméticas

b) Movimientos de Información

c) Operaciones de Entrada y Salida

d) Comparaciones para tomar decisiones

e) Repeticiones de Instrucciones

f) Alteración del Flujo del Proceso

EJEMPLO:

MOVE CLAVE-ARTICULO TO CLAVE-SALIDA.

Esta instrucción indica que se mueva el contenido del Campo de Dato denominado CLAVE-ARTICULO al Campo de Dato denominado CLAVE-SALIDA.

ORACIONES.

Como se mencionó, las Oraciones representan las instrucciones del Programa y siempre comienzan con un Verbo. Las Oraciones se pueden o no terminar con un Punto (Ver ejemplo anterior) pero por lo general se acostumbra terminarlas con Punto. (Existen ciertos casos en donde no se deben terminar con punto, como se verá al describir el verbo IF).

PARRAFOS

En COBOL las Oraciones se estruran en Párrafos. En otras palabras un párrafo es una estructura de oraciones terminadas con un punto que se identifican con un nombre (Nombre de Párrafo).

Los párrafos permiten alterar el flujo de ejecución de las instrucciones o bien para ejecutar un grupo de instrucciones (una vez más veces) sin alterar el flujo de ejecución. Los párrafos se codifican en una sola línea y se terminan con un punto.

EJEMPLO:

```
INICIO.
        OPEN INPUT LECTORA OUTPUT IMPRESORA.
LECTORA.
        READ LECTORA AT END GOTO FIN.
        PERFORM PROCESO.
        GOTO LECTURA.
PROCESO.
FIN.
        CLOSE LECTORA IMPRESORA.
        STOP RUN.
```

TIPOS DE PALABRAS

En el lenguaje COBOL se emplean tres tipos de palabras, que son: Nombres o Sustantivos, Verbos y Palabras Reservadas.

NOMBRES O SUSTANTIVOS

Los nombres se dividen en dos grupos: los definidos por el programador y los proporcionados por el lenguaje COBOL (los cuales tienen ya una función y contenido definido).

Los nombres definidos por el programador se construyen con las mismas reglas empleadas para los Nombres de Datos. Estos se emplean para definir:

Nombres de Datos

Nombres de Registros

Nombres de Archivo

Nombres de Condición

Nombres de Procedimiento o Párrafo

Los nombres que proporciona el lenguaje COBOL son de:

Constantes figurativos

Registros especiales

PALABRAS RESERVADAS

Las palabras reservadas, son palabras en Inglés que tienen un significado ya determinado en el lenguaje COBOL. Estas palabras no se pueden emplear para otro fin diferente para el cual están ya definidas. En la siguiente hoja se proporciona una lista de las palabras reservadas del COBOL.

REGLAS DE PUNTUACION

1. Toda palabra en COBOL se limita a la derecha por lo menos un espacio en blanco.

2. Dos o más nombres de datos escritos en una lista pueden separarse por un espacio en blanco, o por una coma seguida de un espacio en blanco.

HOJA DE CODIFICACION

La forma inicial y la mas común de sumistrar un programa en COBOL a la computadora es empleando tarjetas perforadas (ACTUALMENTE EN DESUSO), las cuales están compuestas de 80 columnas y 12 renglones.

La codificación de los programas en COBOL requieren de un formato ya establecido, el cual se encuentra detallado en estas hojas.

Las hojas de Codificación están compuestas de 80 columnas y 25 renglones (o sea que representan hasta 25 lineas de código) y tienen el siguiente formato:

COLUMNAS	CONTENIDO
1-3	Número de página o hoja. Cada hoja se enumera de 1 en 1 comenzando en 1.
4-6	Número de renglón. Para cada hoja éste se inicia en 10 y se asciende en renglones de 10. El número de página y el número de renglón representan la secuencia de las líneas de código del programa. Ambas secuencias son opcionales.
7	Columna de indicaciones al compilador COBOL. - (guión) Continuación de títulos que no caben en un solo renglón. * (asterisco) Para poner comentarios al programa.
8	Margen A. El COBOL exige que ciertas partes de la codificación principien en esta columna, como son los nombres de las Divisiones, los Párrafos, las secciones, etc.
12	Margen B. Los instrumentos (oraciones) o ciertas declaraciones de la información principian en este margen.
72	Hasta esta columna se puede codificar las declaraciones e instrucciones del Programa.
73-80	Identificación del Programa. Es el nombre que el programador le asigna al programa, el cual puede estar formado de 1 a 8 caracteres. Esta identificación es opcional.

NOMENCLATURA

Un manual de Instrucciones para el lenguaje COBOL muestra al lector las características del lenguaje.

Este texto maneja ciertas reglas de interpretación las cuales son:

a) Palabras Mayúsculas Resaltadas: Deberán usarse como están escritas si queremos usar la función.

b) Palabras Mayúsculas no Resaltadas: Pueden usarse si se desea, sin embargo se emplean para que las declaraciones que escribamos resulten más claras.

c) Palabras Escritas con Minúsculas: Son términos genéricos que indican la clase de palabra que el programador deberá suministrar en la posición mostrada en el formato.

d) Llaves: Cuando hay palabras o frases encerradas entre llaves { } habrá que escoger alguna de ellas.

e) Corchetes: Las palabras encerradas entre [ ] representan opciones.
Lo que contenga puede incluirse en la declaración, si se desea la opción, pero puede omitirse en caso contrario.

f) Puntos Suspensivos (...) : Indican que se puede repetir el concepto que les precede.

NOTA. Las comas (,) o los puntos y comas (;) pueden omitirse en las declaraciones, no así los puntos (.)

EJEMPLOS:

```
MULTIPLY Literal-1      BY identificador-2 [ROUNDED]
         identificador-1
         identificador-3  [ROUNDED]
       [, identificador-4  [ROUNDED] ]...
```

DIVISION DE LOS PROGRAMAS EN COBOL

Todos los programas en COBOL se estructuran en cuatro divisiones, las cuales van en el siguiente orden:

IDENTIFICATION DIVISION

ENVIRONMENT DIVISION

DATA DIVISION

PROCEDURE DIVISION

IDENTIFICATION DIVISION (DIVISION DE IDENTIFICACION)

Consta de unas cuantas líneas, donde se indica: el nombre del programa, el autor, la instalación donde se ejecuta el programa, etc.

ENVIRONMENT DIVISION (DIVISION DE EQUIPO)

Sirve para:

- Especificar las computadoras que se utilizan, tanto para la compilación del programa, como para su ejecución.
- Asignar nombre a los dispositivos de Entrada/Salida que se emplearán en el programa.
- Definir algunas características de los archivos. (Secuencial, Random, etc.)
- Para darles un nombre a ciertas funciones de algunos dispositivos (ejemplo: Salto de hoja en una impresora).

DATA DIVISION (DIVISION DE LOS DATOS)

Se emplea para definir las características de la información que se manejará en el programa. En esta división se realiza:

- Descripción de las características de los archivos de Entrada/Salida.
  - Etiquetas
  - Atributos
  - Nombres de los Resistros
- Descripción de los Resistros de Entrado/Salida
  - Campos en los que se dividen los registros
  - Características de los Campos (tipo y tamaño)
- Descripción de Registros o Campos Auxiliares, como son:
  - Encabezados
  - Contadores
  - Resistros intermedios

PROCEDURE DIVISION (DIVISION DEL PROCESO)

En esta división se define la forma en que se procesará la información que manejará el programa. La División está constituida por oraciones agrupadas por párrafos. La ejecución del programa principia desde el primer párrafo de esta división.

1. IDENTIFICATION DIVISION.

Consta de unas cuantas líneas que identifican el programa.

Nombre del programa, autor, instalación utilizada, fecha de creación y descripción del programa.

La estructura de esta división es como sigue:

```
IDENTIFICATION DIVISION.
    PROGRAM-ID.     Comentario.
   [AUTHOR.         Comentario.]
   [INSTALLATION.   Comentario.]
   [DATE-WRITTEN.   Comentario.]
   [DATE-COMPILED.  Comentario.]
   [SECURITY.       Comentario.]
```

2. ENVIRONMENT DIVISION.

Es la segunda división de un programa en COBOL. Se emplea para definir los dispositivos periféricos que se van a utilizar, asignándoles un nombre de archivo para su manejo. También para definir ciertas características de los archivos, así como para darles nombres a ciertas funciones de algunos dispositivos. La estructura de esta división es como sigue:

```
ENVIRONMENT DIVISION.
 [CONFIGURATION SECTION.
  [SOURCE-COMPUTER. Comentario.]
  [OBJECT-COMPUTER. Comentario.]
  [SPECIAL-NAMES.   Nombres-Especiales.] ]
  [INPUT-OUTPUT SECTION.
  [FILE-CONTROL.
          FILE-CONTROL-DESCRIPTION  .] ]
```

DESCRIPCION:

NOMBRES ESPECIALES:

Control-de-dispositivo-periférico IS Nombre-de-dato

Ejemplo:

```
C01 IS Salta-hoja.
```

FILE-CONTROL-DESCRIPTION.

```
SELECT  Nombre-de-archivo  ASSIGN  TO
        Dispositivo-Periferico.
```

3. DATA DIVISION

La tercera parte de un programa COBOL es la DATA DIVISION, permite la descripción de las características de los archivos de entrada y/o salida, tales características son:

- ETIQUETAS,
- TAMAÑO DEL BLOQUE
- TAMAÑO DE REGISTROS,
- NOMBRE, TAMAÑO Y TIPO DE LOS CAMPOS EN QUE ESTA DIVIDIDO UN REGISTRO
- DESCRIPCION DE REGISTROS O CAMPOS AUXILIARES COMO SON: CONTADORES, ENCABEZADOS, REGISTROS INTERMEDIOS, ETC.

Esta compuesto de varias secciones, las más importantes son:

FILE SECTION Y WORKING-STORAGE SECTION.

La estructura es la siguiente:

```
DATA DIVISION.
[FILE SECTION.
      [FILE-DESCRIPTION
      [RECORD-DESCRIPTION] ...] ...]

[WORKING-STORAGE SECTION.
      [DESCRIPCION-DATO-ELEMENTAL]
      [RECORD-AUXILIAR-DESCRIPTION] ...]
```

DESCRIPCION:

FILE-DESCRIPTION:

```
FD   NOMBRE-DE-ARCHIVO

                          <OMITTED>
     LABEL RECORDS
                          <STANDARD>

     DATA RECORD IS       nombre-de-registro
```

RECORD-DESCRIPTION:

```
                    <Nombre-de-dato>   <Picture>   (A)
Número-de-nivel     <FILLER           >           (X)
                    <F                >  <PIC    >  (9)
                                                  <Literal  >
                                         VALUE
                                                  <Constante>
```

A - Alfabéticos
X - Alfanuméricos
9 - Numéricos

La cláusula VALUE solo puede emplearse en la WORKING-STORAGE SECTION.

4. PROCEDURE DIVISION

PROCEDURE DIVISION.

[nombre-de-sección SECTION.]

nombre-de-parrafo.

[[nombre-de-parrafo.] ... [instrucción.]...] ...]...

LITERALES

Una literal es un elemento del lenguaje COBOL, que representa un valor constante (alfabético, alfanumérico o numérico) dentro de un programa. Existen dos tipos de literales, que son: las literales numéricas y las literales no numéricas.

LITERALES NUMERICAS.

Una literal Numérica, es un elemento formado con los dígitos 0 al 9, los signos + ó - y un punto decimal. De hecho es una cantidad numérica.

Ejemplo:

```
+601     -740     +38.35     ,005
 601      0       74.78      643.
```

El número máximo de dígitos de una literal numérica es de 23.

LITERALES NO NUMERICAS

Una literal no numérica es un elemento formado por cualquiera de los caracteres que maneje el lenguaje COBOL, incluyendo el espacio en blanco. Estas literales deben encerrarse entre comillas (") y el valor de la literal no numérica, los constituyen los caracteres dentro de las comillas. Cualquier par de comillas ("") representan una simple comilla, lo que permite incluirlas en las literales no numéricas. El número de caracteres no debe exceder los 256 caracteres.

Ejemplo:

```
"JOSE JOSE"    "357"    "BC.D"    "47-$"

"B. W. ""BILL""  JONES "    """HOLA"""
```

CONSTANTES FIGURATIVAS

Una constante figurativa es un elemento cuyo valor se encuentra definido en el lenguaje COBOL. Estas constantes se pueden emplear en cualquier parte donde se puede emplear una literal.

TIPOS:

A. ZERO, ZEROS, ZEROES

Representan el valor de cero, o uno ó más caracteres ' cero ', dependiendo para lo que se le emplee.

B. SPACE, SPACES

Representan uno ó más espacios en blanco.

C. QUOTE, QUOTES

Representan uno ó más del caracter " (comilla)

D. ALL literal-no-numérico

Representa la repetición de la literal no numérica, tantas veces como sea necesaria.

ejemplo:

ALL 'ABC' = ABCABC ó ABCAB ó 'AB'

Ejemplos:

```
MOVE SPACE TO LINEA
MULTIPLY CONTADOR BY ZERO
IF CLAVE = ALL '*'
05 NUMERO-DE-HOJA PIC 99 VALUE ZEROS.
```

CLAUSULA PICTURE

[PICTURE] CARACTERES PICTURE.

[PIC]

Sirve para indicar el tamaño (de caracteres) y el tipo (numérico, alfabético ó alfanumérico) de un campo. Si el campo es numérico, sirve para indicar si tiene signo (+ ó -) y si tiene decimal. En la WORKING-STORAGE SECTION permite editar la información para su impresión ($, *, Ø, o, etc).

CARACTERES PICTURE:

C(n)	Indica que el caracteres "C" se repite n veces (ejemplo: 9(5) = 99999).
9	Indica campo numérico
A	Indica campo alfabético
X	Indica campo alfanumérico
V	Indica colocación de punto virtual (no se contabiliza en el tamaño del campo)
S	Indica existencia de signo. (No se contabiliza en el tamaño del campo).
P	Sirve para indicar que el punto decimal está fuera del campo. (No se contabiliza en el tamaño del campo). (max. 18 en 1130)???

ejemplo:

PIC	TAMAMO	CONTENIDO EN MEMORIA	VALOR
VPPP999	3	123	0,000123

CARACTERES PICTURE DE EDICION (sólo WORKING-STORAGE)

Z Indica supresión de ceros no significativos en un campo numérico. Se contabiliza en el tamaño del campo (se colocan a la izquierda) ejemplo:

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
01234	Z2ZZZ	Ø1234
00123	Z(4)9(3)	ØØØØ123
0000	Z(4)	ØØØØ

$ Indica que se debe poner un signo de $ en el campo numérico editado. Se coloca a la izquierda y se contabiliza en el tamaño del campo.

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
1234	$9999	$1234
0023	$$99	Ø$23
23.50	$(2)ZZZ9.99	Ø$ØØ23.50
12	$(4)	Ø$12

- (Signo menos) indica que se ponga un signo menos (-) si el valor del campo es negativo, y si no se pondrá un espacio en blanco. Se debe colocar a la izquierda y se contabiliza en el tamaño del campo. Ejemplos:

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
-1234	----99	-1234
-5	-(3)9	ØØ-5
-5	-ZZ9	-Ø5
-12	----	Ø-12

+ Funciona igual que el signo negativo. Si el valor del campo es negativo se inserta un signo menos.

0 (cero) indica que se coloquen caracteres ceros, sin que se pierdan caracteres del dato fuente. Se contabiliza en el tamaño del campo. Ejemplos:

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
1234	990(2)99	120034
1234	9(4)0(4)	12340000
0012	Z(4)000	ØØ12000

B Indica que se coloquen espacios en blanco, sin que se pierdan caracteres del campo fuente. Ejemplos:

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
1234	99BB99	12ØØ34
1234	9(4)B(2)	1234ØØ

, (Coma) indica que se coloquen comas en el lugar correspondiente, sin que se pierdan caracteres del campo fuente.

. (Punto) indica que se coloque un punto decimal, tomando en cuenta la colocación del punto virtual.
Sólo puede utilizarse en punto decimal.

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
1234	$ZZ,Z(3).99	Ø$1,234.00
-1.23	+9.9(3)	Ø1.230

* (Asterisco) Protección. Indica que se coloquen asteriscos suprimiendo ceros no significativos.

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
1234	***9	1234
1234	*(3)4(9)	***1234
12.00	$*****.99	$***12.00

CR y DB Indica que se coloquen este par de letras a la derecha del campo, siempre que el valor sea negativo, sino dejará los espacios en blanco.

CAMPO FUENTE	PICTURE	RESULTADO IMPRESO
1234	9(4)CR	1234
-1234	9(4)DB	1234DB
-1234	9(4)CR	1234CR

Proposición IF.


```
IF <condición> THEN
    <instrucción(es) a ejecutarse
     cuando la condición se cumpla>
ELSE
    <instrucción(es) a ejecutarse
     cuando la condición no se cumpla>.
NEXT SENTENCE
END-IF
```

Evaluaciones	Operaciones	Condicionales
1) Lógicas	EQUAL LESS GREATER	NOT EQUAL NOT LESS NOT GREATER
2) De clase	NUMERIC ALFABETIC	
3) De condición	nivel 88	asignando valores a una variable dada.

VERBO PERFORM.

PERFORM. Sirve para transferir o ejecutar una secuencia, o serie de secuencias, un número determinado de veces o bien hasta que se satisfaga una cierta condición.

1) PERFORM <nombre de parrafo>
ejecuta las instrucciones desde la etiqueta hasta donde encuentre otra.

2) PERFORM <nombre de párrafo1> THRU <nombre de párrafo2>

Desde hasta

3) PERFORM <nomb-párrafo1> THRU <nomb-párrafo2>
{nomb-dato }
{número entero} TIMES.

ejecuta desde el párrafo1 hasta el párrafo2 tantas veces como nomb-dato o número entero.

4) PERFORM <nomb-párrafo1> THRU <nomb-párrafo2>
UNTIL <condición>.

ejecuta del párrafo1 al párrafo2 hasta que se cumpla la condición.

VERBO MOVE.

```
      { registro-especial }
      { atributo          }
MOVE  { campo-de-dato-1   } TO campo-d-2 [ campo-d-3 ] ...
      { literal-1         }
```

instrucción muy común en el programa fuente; consiste en mover el contenido de campo-de-dato-1 o literal-1, al campo-de-dato-2, sin alterar el contenido del campo-de-dato-1.

Ejemplo:

```
MOVE  RFC  TO  RFC-SAL.
MOVE  ZERO TO  XX,YY,ZZ.
MOVE 'XYZ' TO  NOMBRE.
```

La operación MOVE depende del tipo de dato que se está moviendo.

MOVE ALFANUMERICO:

```
MOVE  ---------------  TO  ---------------
        campo emisor        campo destino
```

: toma el primer caracter izquierdo de la variable emisora y lo lleva a la primera posición de la variable destino; y así sucesivamente todos los demás caracteres, hasta que alguno se termine. Si el emisor tiene un tamaño menor al del destino, los campos faltantes de este último se llenan con blancos.

MOVE NUMERICO:

Campo emisor ._._._._.

Campo destino ._0._._._.

: toma el primer caracter apartir del punto y mueve de derecha a izquierda.

Por lo tanto se concluse que si:

C. Emisor > C. Destino
- Se pierden cifras más significativas.

C. Emisor < C. Destino
- Llena de Ceros.

NOTA: Si el punto no está especificado lo toma por default como primera posición de lo izquierda.

Ejemplo:

C. Receptor definido como 999V99

5 4 3 2 1 6 7 8 9 0

C. Emisor C. Receptor 3 2 1 0 0

C. Receptor después del movimiento.

C. Emisor definido como 99V999 2 1 0 0 0

C. Receptor definido como 999999 5 4 3 2 1

1 0 0 1 6 2 1 3 7 4 9 0 C. Receptor después del mov.

Emisor Receptor 0 0 0 0 1 0

Existen situaciones en las cuales hay que mover la mayor parte del contenido de un registro a campos de datos del mismo nombre en otro registro, siendo gravoso escribir todos los MOVE necesarios para tales casos.

La opción CORRESPONDING del verbo MOVE hace esto innecesario.

MOVE CORRESPONDING nombre-dato-1 TO nombre-dato-2.

Características:

- Ambos nombres de dato deben corresponder a niveles de grupo.
- Se mueven todos los campos dentro de nombre-dato-1 que tengan el mismo nombre que otro campo dentro de nombre-dato-2, tal y como si hubiéramos escrito una serie de MOVE sencillos.

Ejemplo:

MOVE A in Emisor TO A in Receptor

MOVE E in Emisor TO E in Receptor

MOVE C in Emisor to C in Receptor

MOVE D in Emisor to D in Receptor

MOVE F in Emisor to F in Receptor

A B C D E F G H Emisor

A D E B F G H C Receptor

Considerando que Emisor y Receptor son nombres de registros y tienen el formato mostrado:

el efecto total de

MOVE CORRESPONDING Emisor TO Receptor

sería exactamente el mismo.

VERBO ADD.

```
                ( identificador-1 ) [ ( identificador-2 ) ]
ADD             ( literal-1       )   ( literal-2       )  ... TO
      identificador-m [ROUNDED ]
   [, identificador-n ROUNDED] ...
      [ ON SIZE ERROR instrucción ELSE instrucción ]
```

NOTA: El otro formato de la proposición ADD es exactamente igual, sólo cambiando la palabra TO por la palabra GIVING.

Las palabras TO y GIVING no pueden usarse ambas en la misma oración.

ADD

Para poder usar este verbo en una oración es necesario contar por lo menos, con dos operandos numéricos, el o los sumandos y el operando sobre el que se almacenará el resultado.

Si la opción escogida usa la palabra TO:

El resultado de la suma reemplazará el contenido original del campo que le sigue a la palabra TO.

Ejemplo:

ADD Area TO Manejo.

Si en el resultado de la operación el número de dígitos decimales excede al espacio disponible en el campo asignado al resultado, perderán los dígitos a menos que se use seguido de la palabra ROUNDED (que redondeará la cifra al espacio especificado).

Cuando el número total de cifras a la izquierdo del punto decimal excede al espacio disponible en el campo-dato asignado al resultado, ocurrirían cosas impredecibles si no se usa ON SIZE ERROR GO TO etiqueta-2 (prevención al error de tamaño).

Ejemplo:

```
ADD Area, Manejo, Caja GIVING Total ROUNDED, ON SIZE
ERROR GO TO Rutina-0.
```

Cuando se desea sumar dos o más cantidades y hacer algún otro campo de datos igual a la suma, se usa GIVING.

```
VERBO SUBTRACT.
        {literal-1       } [ {literal-2       } ]
SUBTRACT { nombre-dato-1 } [,{ nombre-dato-2 }] ... FROM

 { literal-N       }     [ GIVING nombre-de-dato-m ]
 { nombre-de-dato-N }

 [ROUNDED]
 [; ON SIZE ERROR instrucción [ELSE instrucción]]
```

NOTA: El otro formato del verbo SUBTRACT es exactamente igual pero sin la palabra GIVING.

Si se desea restar un campo de datos de otro y guardar la diferencia en un tercer campo, debe usarse la opción GIVING.

Todos los operandos antes de la palabra FROM significa que es posible formar la suma de varios operandos y restar esta suma del operando de la derecha de la palabra FROM.

Ejemplo:

```
SUBTRACT A,B, FROM C GIVING D.
```

	A	B	C	D
ANTES	1	3	5	29
DESPUES	1	3	5	01

Si no se usa GIVING, el resultado de la diferencia reemplazará al minuendo del campo que sigue a la palabra FROM.

```
SUBTRACT AREA-1, AREA-2, AREA-3 FROM AREA-4 ROUNDED,
ON SIZE ERROR GO TO RUTINA-3.
```

VERBO MULTIPLY.

```
                {nombre-dato-1 }       {  nombre-de-dato-2}
MULTIPLY        { literal-1    }  BY    { literal-2        }

[GIVING nombre-de-dato-3] [ROUNDED]

[; ON SIZE ERROR instrucción [ELSE instrucción]]
```

NOTA: Si no se usa GIVING, será reemplazado el producto en el multiplicando (segundo factor).

MULTIPLY.

Este verbo aritmético multiplica dos cantidades y hace una segunda o una tercera variable igual al producto.

Ejemplo:

```
MULTIPLY Piezas BY Costo GIVING Precio.
          PIEZAS    COSTO     PRECIO
ANTES       23        147      4444444
DESPUES     23        147      0003381
```

VERBO DIVIDE.

```
          {  literal-1       }       {  literal-2       }
DIVIDE    { identificador-1  } INTO  { identificador-2  }

[GIVING identificador-3] [ROUNDED] [REMAINDER

                identificador-4 ROUNDED]

[;ON SIZE ERROR instrucción [ ELSE instrucción ]]
```

NOTA: Si no se usa GIVING, será reemplazado el cociente en el dividendo por lo tanto, este último operando no podrá ser una literal.

Ejemplo:

```
DIVIDE A INTO B GIVING C ROUNDED.
ON SIZE ERROR GO TO RUTINA-4
```

(Divide B entre A, dando C redondeado, en caso de error de tamaño ir a rutina-4).

VERBO COMPUTE.

```
COMPUTE  nom-dato-1  [ROUNDED] [,nom-dato-2  ROUNDED] ...
         { FROM    }
         { =       } { nombre-de-dato }
         { EQUALS  } { fórmula        }

[; ON SIZE ERROR instrucción [ELSE instrucción]]

 COMPUTE.
```

Este verbo ofrece una forma mucho más compacta de especificar operaciones aritméticas.

Si cualquier elemento en la fórmula es un nombre-dato, deberá definirse en la División de Datos.

Ejemplo:

```
COMPUTE X = A + (B * C)
```

ó también:

```
COMPUTE X ROUNDED = A + (B * C) ON SIZE ERROR GO TO
RUTINA-1
```

NOTA: Para expresiones aritméticas, las reglas de jerarquía y ejecución son las mismas que el álgebra común.

VERBO INSPECT.

Formato:

```
INSPECT nom-campo-datos TALLYING
                          ALL
    literal-1 FOR         LEADING        literal-2
                          CHARACTERS
    BEFORE
                  INITIAL   literal-3
    AFTER

                  CHARACTERS BY
    REPLACING
                  ALL
                  LEADING   literal-4   BY   literal-5
                  FIRST

    BEFORE
                  INITIAL   literal-6
    AFTER
```

TALLYING	- Contabilizando en literal-1
ALL	- TODOS
BEFORE	- ANTES
AFTER	- DESPUES
FIRST	- PRIMER
LEADING	- AL PRINCIPIO (ENCABEZANDO)
CHARACTERS	- CARACTERES

INSPECT examina un campo de datos de izquierda a derecha, realizando:

1. Conteo del número de veces que aparece un caracter

específico.

2. Reemplazo de caracteres.

3. Conteo y reemplazo (incisos 1 y 2).

EJEMPLOS

	nom-campo-datos antes	nom-campo-datos después	literal-1
INSPECT nom-campo-datos TALLYING literal-1 FOR LEADING "L"	LULU	LULU	1
	HOLA	HOLA	0
	LLORA	LLORA	2
INSPECT nom-campo-datos TALLYING literal-1 FOR ALL "L"	LULU	LULU	2
	HOLA	HOLA	1
	LLORA	LLORA	2
INSPECT nom-campo-datos REPLACING ALL "M" BY "P"	MAMA	PAPA	
	MAMUT	PAPUT	2
INSPECT nom-campo-datos TALLYING literal-1 FOR CHARACTERS AFTER "S" REPLACING ALL "I" BY "O"	MISSISSIPI	MOSSOSSOPO	7
	HISTORIA	HOSTOROA	5
INSPECT nom-campo-datos REPLACING ALL "I" BY "O" ALL "S" BY "Z" AFTER "MIS"	MISSISSIPI	MOSZOZZOPO	
	HEMISFERIO	HEMOSFEROO	

REDEFINES.

Esta cláusula permite que una misma área de almacenamiento en memoria (registro, parte de un registro o un simple campo) se puede referenciar con más de un nombre con diferente formato.

El formato de esta cláusula es:

Número-de-nivel Nombre-dato-1 REDEFINES Nombre-dato-2

Donde:

Nombre-dato-2 es el nombre-de-dato que describe el formato del area de almacenamiento original.

Nombre-dato-1 es el nombre-de-dato que describe un nuevo formato para el area de almacenamiento original referenciada por nombre-dato-2.

REGLAS

1. La cláusula sólo se permite para redefinir parte de un registro en la FILE SECTION (o sea que no se puede emplear con el número de nivel 01), ya que la redefinición de registros en esta sección es indicada en la cláusula DATA RECORDS.

2. La cláusula si permite redefinir todo o parte de un registro en la WORKING-STORAGE SECTION.

3. Los números de nivel del nombre-de-dato-1 y el nombre-de-dato-2 deben ser idénticos y no podrán ser 66 u 88.

4. Las áreas referenciadas por los nombres de dato deberán tener el mismo tamaño, aunque podrán tener diferente formato.

5. La nueva descripción de una área original que se redefine, debe seguir a continuación de la descripción de esta última, sin ninguna otra descripción intermedia.

6. Más de una redefinición a una área original es permitida. Las nuevas redefiniciones a una área original deberán hacerse a esta última únicamente (o sea que el nombre-de-dato-2 deberá ser el mismo en todas las nuevas redefiniciones del área que representa el nombre-de-dato-2).

7. La cláusula VALUE sólo es permitida en el campo que originalmente define el área y no es permitida en los campos que redefinen al campo en cuestión. (Con excepción de los niveles 88).

8. Un campo con nivel 01 que emplea la cláusula OCCURS no debe contener la cláusula REDEFINES.

EJEMPLOS:

```
01  REGISTRO-1.
    02  PARTE-UNO              PIC X(60).
    02  PARTE-DOS REDEFINES PARTE-UNO.
        03  XX                 PIC X(40).
        03  YY                 PIC 9(20).
    02  PARTE-TRES REDEFINES PARTE-UNO.
        03  ZZ                 PIC A(55).
        03  FILLER             PIC X(5).
01  TIPO-DE-ARTICULOS.
        03  ARTICULO-1         PIC XX VALUE "A"
        03  ARTICULO-2         PIC X) VALUE "B"
        03  ARTICULO-3         PIC XX VALUE "?"
01  TABLA-ARTICULOS REDEFINES TIPO-DE-AN
        03  ARTICULOS          PIC XX OCCURRS 3
```

CLAUSULA OCCURS

Esta Cláusula permite definir tablas. Una tabla es un conjunto de campos de información donde cada campo tiene las mismas características (tamaño y formato).

El formato de esta Cláusula tienen dos formas:

FORMA 1:

```
{OC    . }
{OCCURS}   entero-2  TIMES

[{ASCENDING  }
[{DESCENDING }  KEY IS nom-dato-2 [, nom-dato-3] ... ] ...]

[INDEXED BY  nombre-indice-1  [,nombre-indice-2] ...]
```

FORMA 2:

```
{OC     }
{OCCURS }  entero-1 TO  entero-2  TIMES

        [DEPENDING ON nombre-dato-1]

[ [ {ASCENDING }
[ [ {DESCENDING} KEY IS nom-dato-2 [,nom-dato-3] ...] ...]

[INDEXED BY   nombre-indice-1  [,nombre-indice-2 ] ...]
```

REGLAS:

1. El entero-1 y el entero-2 tienen que ser números enteros positivos.

2. El entero-1 debe ser menor que el entero-2

3. El contenido del nombre-dato-1 debe contener un valor entero positivo; este define el tamaño máximo de la tabla, pero no debe exceder el valor del entero-2.

4. El nombre de dato-2 puede ser el nombre del campo que contiene la cláusula OCCURS ó un campo en que se divide el primero.

5. El nombre-dato-3, etc. sólo puede ser un campo en que se divide el campo que emplee la cláusula OCCURS.

6. La cláusula OCCURS no puede emplearse en campos que tengan los siguientes números de nivel: 77,66 y 88.

7. Para hacer referencia a un campo que emplea la cláusula OCCURS y a los campos en que se divide éste primero, se hace utilizando indices (enteros, nombres de datos o nombres de indice). Los indices se encierran entre parentesis y deben seguir al nombre del campo.

8. La cláusula OCCURS se puede emplear en cualquier sección de la DATA DIVISION.

9. Los indices no deben utilizarse al emplear el verbo SEARCH.

10. En la forma-1 el valor del entero-2 determina el tamaño maximo de la tabla.

11. En la forma-2, el valor del entero-1 define el tamaño minimo de la tabla y el entero-2 define el tamaño máximo de la tabla. El tamaño exacto de la tabla lo determina el valor del contenido del nombre-de-dato-1.

12. La opción KEY IS sirve para indicar que la tabla se encuentra organizada ascendente ó Descendente con respecto a los nombres de dato 2,3,etc. Estos se escriben en orden descendente de acuerdo a su prioridad.

13. La opción INDEXED BY sirve para asociarle uno ó más indices a la tabla. Estas dos últimas cláusulas son requeridas para emplear el verbo SEARCH.

Ejemplos prácticos en

COBOL

```
columna
   8   12

    IDENTIFICATION DIVISION.
    PROGRAM-ID. NOMINITA.
    AUTHOR.
    DATE-WRITTEN.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. VAX 11-780.
    OBJECT-COMPUTER. VAX 11-780.
    SPECIAL-NAMES. CHANNEL 1 IS BRINCO.
    INPUT-OUTPUT SECTION
    FILE-CONTROL.
        SELECT ARCHIVO ASSING TO READER.
        SELECT IMPRESOR ASSIGN TO PRINTER.
    DATA DIVISION.
    FILE SECTION.
    FD  ARCHIVO
        LABEL RECORD STANDARD
        DATA RECORD IS DATOS.
    01  DATOS.
        03 NOMBRE     PIC X(30).
        03 SUELDO-DIARIO    PIC 9(4)V99.
        03 DIAS-TRABAJADOS  PIC 99.
        03 SEXO  PIC X.
    FD  IMPRESOR
        LABEL RECORD STANDARD
        DATA RECORD IS LINEA.
    01  LINEA  PIC X(132).
    WORKING-STORAGE SECTION.
    77  A1   PIC 9(6)V99.
    77  SUELDO-MENSUAL PIC 9(6)V99.
    77  NUM-LINEAS     PIC 9(4)  VALUE 60.
    01  TITULO.
        03 FILLER  PIC X(15)  VALUE SPACES.
        03 FILLER  PIC X(6)   VALUE "NOMBRE".
        03 FILLER  PIC X(29)  VALUE SPACES.
        03 FILLER  PIC X(14)  "SUELDO MENSUAL".
        03 FILLER  PIC X(68)  VALUE SPACES.
    01  DETALLE.
        03 FILLER  PIC X(10)  VALUE SPACES.
        03 NOMBRE-LISTADO  PIC X(30).
        03 FILLER  PIC X(10)  VALUE SPACES.
        03 SUELDO-MENSUAL-LIS  PIC   ZZZ,ZZ9.99.
        03 FILLER  PIC X(71)  VALUE SPACES.
    PROCEDURE DIVISION.
    ABRIR-LEER.
        OPEN INPUT ARCHIVO
             OUTPUT IMPRESOR.
        READ ARCHIVO
```

```
INICIO.
    PERFORM CALCULOS.
    PERFORM ESCRITO.
    READ ARCHIVO
        AT END GO TO TERMINO.
        GO TO INICIO.
    CALCULOS.
        MULTIPLY SUELDO-DIARIO  BY DIAS-TRABAJADOS
            GIVING SUELDO-MENSUAL.
        IF SEXO = "M"
            PERFORM 5-POR-CIENTO
        ELSE
            PERFORM 3-POR-CIENTO.
    5-POR-CIENTO.
        MULTIPLY 0.05 BY SUELDO-MENSUAL GIVING A1.
        ADD A1 TO SUELDO-MENSUAL.
    3-POR-CIENTO.
        MULTIPLY 0.03 BY SUELDO-MENSUAL  GIVING A1
        ADD A1 SUELDO-MENSUAL.
    ESCRIBO.
        MOVE NOMBRE TO NOMBRE-LISTADO
        MOVE SUELDO-MENSUAL TO SUELDO-MENSUAL-LIS.
        IF NUM-LINEAS > 57
            MOVE 4 TO NUM-LINEAS
            WRITE LINEA FROM TITULO AFTER BRINCO
        ELSE
            ADD 3 TO NUM-LINEAS.
            WRITE LINEA FROM DETALLE AFTER 3.
    TERMINO.
        CLOSE ARCHIVO
              IMPRESOR
              STOP RUN.

  datos.   DATA ARCHIVO

nombre...30 columnas.....nnnnnnnddx

JORGE GARCIA CAMACHO              02503330
JOSE LUIS IGLESIAS                 2282028
ANDREA DORIA                       4507030M
JULIETA ROMERO                     6359029M
SOCORRO ALBINO                     6778031M
DOROTEO ARRANCO                    3189030
IVAN IVENIAN                       4500030

nombre.............................sueldo...d.s
                                             1.0
 dias  2 columnas.                           x.x
 sexo  1 columna                             s.0
```

```
*****************************************************************
***                                                           ***
***          ESTE PROGRAMA CREA ARCHIVOS DE EMPLEADOS         ***
***          LAS ESPECIFICACIONES DE DICHO ARCHIVOS SON:      ***
***                        ESnn.DAT                           ***
***            DONDE:                                         ***
***                        E = EMPLEADOS.                     ***
***                        S = SEMANA.                        ***
***                       nn = NUMERO DE LA SEMANA.           ***
***                                                           ***
*****************************************************************
*
IDENTIFICATION DIVISION.
PROGRAM-ID.                ALTA.
AUTHOR.                    GUSTAVO Y NICOLAS.
INSTALLATION.              AQUI EN EL CECAFI.
DATE-WRITTEN.              OCTUBRE DE 1985.
DATE-COMPILED.             ESTE DIA.
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
*
*********************************************
**    LA ENTRADA DE DATOS ES POR TERMINAL.   **
*********************************************
*
    SELECT DATOS-TERM ASSIGN TO SYS$INPUT.
*
*********************************************
**    EL NUMERO DE LA SEMANA DETERMINA EL    **
**          NOMBRE DEL ARCHIVO.              **
*********************************************
*
    SELECT EMPLEADOS ASSIGN TO ".DAT".
*
DATA DIVISION.
FILE SECTION.
FD EMPLEADOS
    LABEL RECORDS ARE STANDARD
    VALUE OF ID IS CLAVE-EMPLEADOS
    DATA RECORD IS REGISTRO-EMPLEADOS.
01  REGISTRO-EMPLEADOS.
    03 FICHA-EMPLEADOS          PIC 9(05).
    03 NOMBRE-EMPLEADOS         PIC X(30).
    03 SUELDO-EMPLEADOS         PIC 9(06)V99.
FD  DATOS-TERM
    DATA RECORD IS REGISTRO-TERMINAL.
01  REGISTRO-TERMINAL           PIC X(80).
WORKING-STORAGE SECTION.
*
*************************************************
```

```
**    DEFINICION DEL LA IDENTIFICACION DEL ARCHIVO **
**                  DE SALIDA.                      **
*****************************************************
*
01   CLAVE-EMPLEADOS.
     03 FILLER                  PIC X(02) VALUE "ES".
     03 SEM-CLAVE-EMPLEADOS     PIC 9(02) VALUE ZEROES.
77   EDO-TERMINAL               PIC 9(01) VALUE ZERO.
     88 FIN-TERMINAL VALUE 1.
PROCEDURE DIVISION.
INICIO.
     DISPLAY "DAME EL  DE LA SEMANA (dos disitos): " WITH NO
ADVANCING.
     ACCEPT SEM-CLAVE-EMPLEADOS.
     OPEN OUTPUT EMPLEADOS.
     OPEN INPUT DATOS-TERM.
     PERFORM ALTAS UNTIL FIN-TERMINAL.
     CLOSE EMPLEADOS DATOS-TERM.
     STOP RUN.
ALTAS.
     DISPLAY "    FICHA          NOMBRE              SUELDO".
     DISPLAY "    99999X                            X99999".
     DISPLAY "    " WITH NO ADVANCING.
     MOVE SPACES TO REGISTRO-TERMINAL
     READ DATOS-TERM NEXT RECORD INTO REGISTRO-EMPLEADOS
        AT END MOVE SPACES TO REGISTRO-EMPLEADOS.
     IF NOMBRE-EMPLEADOS = SPACES THEN
        MOVE 1 TO EDO-TERMINAL
     ELSE
        WRITE REGISTRO-EMPLEADOS.
```

```
Identification Division.
Program-id.     Solucion-de-la-ec-de-2do-grado.
Author.         Instructores de Cobol.
Installation.   Cecafi.
Date-written.   Octubre 1995.
Security.       Ninguna.
*
Environment division.
Configuration Section.
Source-computer. Digital-VAX-11-780.
Object-computer. Digital-VAX-11-780.
Input-output section.
File-control.
    Select datos assign to sys$input.
    Select salida assign to sys$output.
*
Data division.
File section.
Fd   datos
     Data record is Datos1.
01   Datos1.
       03 A picture is s9(02)v9(02) sign is leading separate character.
       03 B picture is s9(02)v9(02) sign is leading separate character.
       03 C picture is s9(02)v9(02) sign is leading separate character.
Fd  salida
    Data record is linea.
01  Linea picture is x(80).
Working-storage section.
77  Resultado picture is s9(02)v9(02) value zeroes.
77  Disc      picture is s9(02)v9(02).
77  X1        picture is s9(02)v9(02).
77  X2        picture is s9(02)v9(02).
01 Linea-aux.
       03 Filler picture is x(16) value "el resultado es:".
       03 Filler picture is x(03) value all ": ".
       03 Result picture is +9(02).9(02).
       03 Filler picture is x(55) value all " ".
*
Procedure division.
Inicio.
    Open input datos.
    Open output salida.
    Display " "
    Display "            Solucion de la ecuacion cuadratica."
    Display "            ***********************************"
    Display " "
    Display "  3 datos en linea, 1: a,  2: b,  3: c"
    Display "                    s : signo     eedd: valor, sin punto
decimal"
    Display "                                  ee  enteros"
    Display "                                  dd  decimales"
    Display " "
```

```
    Display "SeeddSeeddSeedd"
    Read datos at end display " No hubo datos." stop run.
    If a is equal to zero and b is equal to zero and c is equal to 0
stop run.
    If a is equal to zero and b is equal to zero stop run.
    If a is equal to zero and c is equal to zero stop run.
    If b is equal to zero and c is equal to zero stop run.
    If a is equal zero then
        display "Ecuacion de primer grado."
        compute x1 = ((-1 * c)/b)
        display "x1 = " x1
        move x1 to result
        write linea from linea-aux
    else
        compute disc = (b ** 2 - 4 * a * c).
    if disc is less than zero
        display "Raices complejas no puedo resolverlo."
    else
        display "Raices reales."
        compute x1 = ((-b - disc ** 0.5)/(2*a))
        compute x2 = ((-b + disc ** 0.5)/(2*a))
        move x1 to result
        write linea from linea-aux
        move x2 to result
        write linea from linea-aux.
    Stop run.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ETIQUETAS.
AUTHOR. JORGE VALERIO.
INSTALLATION. CECAFI.
SECURITY. PROGRAMA QUE EMITE ETIQUETAS DE UN ARCHIVO DE ENTRADA
          A UN ARCHIVO DE SALIDA.
**************************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX-11-780.
OBJECT-COMPUTER. VAX-11-780.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATOS-DE-ENTRADA ASSIGN TO
'[CURSOS.COBOL.CURCOBOL]NOMBRES.DAT'.
    SELECT ENVIANDO-ETIQUETAS ASSIGN TO
'[CURSOS.COBOL.CURCOBOL]ETIQUETAS.LIS'.
**************************************************************************
DATA DIVISION.
FILE SECTION.
FD  DATOS-DE-ENTRADA
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS TARJETA.
01  TARJETA.
    02 NOMBRE PICTURE IS X(25).
    02 CALLE PICTURE IS X(25).
    02 CIUDAD PICTURE IS X(30).
*
FD  ENVIANDO-ETIQUETAS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS REGISTRO-DE-IMPRESION.
01  REGISTRO-DE-IMPRESION.
    02 LINEA-DE-IMPRESION PICTURE IS X(30).
*
WORKING-STORAGE SECTION.
01  INDICADOR-DE-FIN-DE-DATOS PICTURE IS X(02).
**************************************************************************
PROCEDURE DIVISION.
PROGRAMA-PRINCIPAL.
    OPEN INPUT DATOS-DE-ENTRADA OUTPUT ENVIANDO-ETIQUETAS.
    MOVE "NO" TO INDICADOR-DE-FIN-DE-DATOS.
    PERFORM LECTURA-DE-TARJETAS.
    PERFORM LECTURA-E-IMPRESION UNTIL INDICADOR-DE-FIN-DE-DATOS IS
EQUAL TO "SI".
    CLOSE DATOS-DE-ENTRADA ENVIANDO-ETIQUETAS.
    STOP RUN.
*
LECTURA-DE-TARJETAS.
    READ DATOS-DE-ENTRADA RECORD AT END MOVE "SI" TO
INDICADOR-DE-FIN-DE-DATOS.
*
LECTURA-E-IMPRESION.
```

```
MOVE NOMBRE TO LINEA-DE-IMPRESION.
WRITE REGISTRO-DE-IMPRESION BEFORE ADVANCING 1 LINE.
MOVE CALLE TO LINEA-DE-IMPRESION.
WRITE REGISTRO-DE-IMPRESION BEFORE ADVANCING 1 LINE.
MOVE CIUDAD TO LINEA-DE-IMPRESION.
WRITE REGISTRO-DE-IMPRESION BEFORE ADVANCING 4 LINE.
PERFORM LECTURA-DE-TARJETAS.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EJEMPLO-DE-ARCHIVO-INDEXADO.
AUTHOR. LAURA Y SERGIO SANDOVAL.
*
*
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
    SOURCE-COMPUTER. EN-VAX-11.
    OBJECT-COMPUTER. EN-VAX-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SALIDA ASSIGN TO "SALIDA.DAT"
           ORGANIZATION IS INDEXED.
    SELECT ENTRADA ASSIGN TO "SYS$INPUT".
*
*
DATA DIVISION.
FILE SECTION.
FD  SALIDA
    DATA RECORD IS REG-SAL
    ACCESS MODE DYNAMIC
    RECORD KEY IS NUMCTA-S.
01  REG-SAL.
    03 NUMCTA-S        PIC 9(08).
    03 NOMBRE-S        PIC X(32).
    03 PROMEDIO        PIC 99V99.
FD  ENTRADA
    DATA RECORD IS REG-ENT.
01  REG-ENT.
    03 NUMCTA-E        PIC 9(08).
    03 NOMBRE-E        PIC X(32).
    03 PROMEDIO        PIC 99V99.
*
WORKING-STORAGE SECTION.
*
*
PROCEDURE DIVISION.
PARRAFO-INICIO.
    OPEN INPUT ENTRADA.
    OPEN I-O SALIDA.
    READ ENTRADA AT END
        DISPLAY "YA NO HAY DATOS"
    END-READ.
    PERFORM CREA-INDEXADO UNTIL NUMCTA-E=99999999.
    DISPLAY "DAME EL NUMERO DE CUENTA A BUSCAR".
    ACCEPT NUMCTA-S.
    PERFORM BUSQUEDA UNTIL NUMCTA-S=99999999.
    DISPLAY "DAME NUMCTA PARA MODIFICAR NOMBRE"
    ACCEPT NUMCTA-S.
    PERFORM MODIFICA UNTIL NUMCTA-S=99999999.
    DISPLAY "DAME NUMCTA PARA BORRAR EL REGISTRO"
    ACCEPT NUMCTA-S.
```

```
    PERFORM BORRA UNTIL NUMCTA-S=99999999.
    STOP RUN.
CREA-INDEXADO.
    MOVE REG-ENT TO REG-SAL.
    WRITE REG-SAL INVALID KEY
        DISPLAY "ERROR EN LLAVE" NUMCTA-E
    END-WRITE.
    READ ENTRADA AT END
        DISPLAY "FIN DE DATOS"
    END-READ.
BUSQUEDA.
    READ SALIDA KEY IS NUMCTA-S
        INVALID KEY DISPLAY "NO ENCONTRO NUMCTA"
    END-READ.
    DISPLAY "NUMCTA " NUMCTA-S "NOMBRE " NOMBRE-S.
    DISPLAY "DAME EL NUMERO DE CUENTA A BUSCAR".
    ACCEPT NUMCTA-S.
MODIFICA.
    START SALIDA KEY IS EQUAL NUMCTA-S INVALID KEY
        DISPLAY "NO HAY REGISTRO A MODIFICAR"
    END-START.
    DISPLAY "NOMBRE A MODIFICAR".
    ACCEPT NOMBRE-S.
    REWRITE REG-SAL INVALID KEY
        DISPLAY "SE SE HIZO EL REWRITE"
    END-REWRITE.
    DISPLAY "DAME NUMCTA PARA MODIFICAR NOMBRE"
    ACCEPT NUMCTA-S.
BORRA.
    READ SALIDA KEY IS NUMCTA-S INVALID KEY
        DISPLAY "NO HAY REGISTRO A BORRAR"
    END-READ.
    DELETE SALIDA INVALID KEY
        DISPLAY "NO SE EFECTUO EL DELETE"
    END-DELETE.
    DISPLAY "DAME NUMCTA PARA BORRAR EL REGISTRO"
    ACCEPT NUMCTA-S.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MARIAS.
AUTHOR. FREBECARIOS.
INSTALLATION. VAX 11-780.
DATE-WRITTEN. 22 FEBRERO 1985.
DATE-COMPILED. 22 FEBRERO 1985.
SECURITY. NINGUNA.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATOS ASSIGN "MARIAS.DAT"
    ORGANIZATION IS SEQUENTIAL.
    SELECT SALIDA ASSIGN "REPORTE.DAT"
    ORGANIZATION IS SEQUENTIAL.
    SELECT SALIDA-NOVEDADES ASSIGN "NOVEDADES.DAT"
    ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD DATOS
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS REGISTRO.
01 REGISTRO.
    03 NOMBRE PIC X(20).
    03 LOCALIZACION PIC X(20).
    03 ARTICULO PIC X(20).
    03 PRECIO-UNITARIO PIC 9(03).
    03 CANTIDAD PIC 9(03).
    03 FILLER PIC X(14).
FD SALIDA
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS DESCRIPCION-SALIDA.
01 DESCRIPCION-SALIDA.
    03 LINEA-DE-IMPRESION PIC X(80).
FD SALIDA-NOVEDADES
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS DESCRIPCION-SALIDA-NOV.
01 DESCRIPCION-SALIDA-NOV.
    03 LINEA-DE-IMPRESION PIC X(80).
WORKING-STORAGE SECTION.
77 FIN-DE-ARCHIVO PIC X(02) VALUE 'NO'.
77 TOTAL-DIARIO PIC 9(06) VALUE ZERO.
77 TOTAL-GLOBAL PIC 9(09) VALUE ZEROES.
77 LINEA-DE-BLANCOS PIC X(80) VALUE SPACES.
01 REGISTRO-AUXILIAR-1.
    03 FILLER PIC X(09) VALUE SPACES.
    03 FILLER PIC X(06) VALUE 'NOMBRE'.
    03 FILLER PIC X(13) VALUE SPACES.
    03 FILLER PIC X(12) VALUE 'LOCALIZACION'.
    03 FILLER PIC X(12) VALUE SPACES.
    03 FILLER PIC X(08) VALUE 'ARTICULO'.
    03 FILLER PIC X(10) VALUE SPACES.
    03 FILLER PIC X(05) VALUE 'TOTAL'.
```

```
    03 FILLER PIC X(05) VALUE SPACES.
01 REGISTRO-AUXILIAR-2.
    03 FILLER PIC X(02) VALUE SPACES.
    03 NOMBRE PIC X(20).
    03 FILLER PIC X(02) VALUE SPACES.
    03 LOCALIZACION PIC X(20).
    03 FILLER PIC X(02) VALUE SPACES.
    03 ARTICULO PIC X(20).
    03 FILLER PIC X(04) VALUE SPACES.
    03 TOTAL PIC 9(06).
    03 FILLER PIC X(04) VALUE SPACES.
01 REGISTRO-AUXILIAR-3.
    03 FILLER PIC X(02) VALUE SPACES.
    03 FILLER PIC X(20) VALUE 'REPORTE DE NOVEDADES'.
    03 FILLER PIC X(58) VALUE SPACES.
01 REGISTRO-AUXILIAR-4.
    03 FILLER PIC X(05) VALUE SPACES.
    03 FILLER PIC X(08) VALUE 'HAY QUE '.
    03 AYUDAR-CAMBIAR PIC X(07).
    03 FILLER PIC X(03) VALUE ' A '.
    03 NOMBRE PIC X(20).
    03 FILLER PIC X(37) VALUE SPACES.
01 REGISTRO-AUXILIAR-5.
    03 FILLER PIC X(05) VALUE SPACES.
    03 FILLER PIC X(15) VALUE 'ESTA VENDIENDO '.
    03 ARTICULO PIC X(20).
    03 FILLER PIC X(03) VALUE 'EN '.
    03 LOCALIZACION PIC X(20).
    03 FILLER PIC X(17) VALUE SPACES.
01 REGISTRO-AUXILIAR-6.
    03 FILLER PIC X(60) VALUE SPACES.
    03 FILLER PIC X(06) VALUE 'TOTAL:'.
    03 TOTAL PIC 9(09).
    03 FILLER PIC X(04) VALUE SPACES.
PROCEDURE DIVISION.
PRINCIPAL.
    PERFORM PROCESO-1.
    MOVE 'NO' TO FIN-DE-ARCHIVO
    PERFORM PROCESO-2.
    STOP RUN.
PROCESO-1.
    OPEN INPUT DATOS.
    OPEN OUTPUT SALIDA.
    PERFORM PARRAFO-1.
    PERFORM PROCESO-REPORTE-DIARIO UNTIL FIN-DE-ARCHIVO EQUAL "SI".
    MOVE TOTAL-GLOBAL TO TOTAL IN REGISTRO-AUXILIAR-6.
    WRITE DESCRIPCION-SALIDA FROM REGISTRO-AUXILIAR-6 AFTER ADVANCING
5 LINES.
    CLOSE DATOS.
    CLOSE SALIDA.
PROCESO-REPORTE-DIARIO.
    READ DATOS AT END MOVE "SI" TO FIN-DE-ARCHIVO.
    IF FIN-DE-ARCHIVO NOT EQUAL "SI" THEN
```

```
        PERFORM PARRAFO-2.
*   NOELSE
*   ENDIF
PARRAFO-1.
    WRITE DESCRIPCION-SALIDA FROM REGISTRO-AUXILIAR-1 AFTER ADVANCING
5 LINES.
PARRAFO-2.
    MULTIPLY CANTIDAD BY PRECIO-UNITARIO GIVING TOTAL-DIARIO.
    ADD TOTAL-DIARIO TO  TOTAL-GLOBAL
    MOVE NOMBRE IN REGISTRO TO NOMBRE IN REGISTRO-AUXILIAR-2.
    MOVE LOCALIZACION IN REGISTRO TO LOCALIZACION IN
REGISTRO-AUXILIAR-2.
    MOVE ARTICULO IN REGISTRO TO ARTICULO IN REGISTRO-AUXILIAR-2.
    MOVE TOTAL-DIARIO TO TOTAL IN REGISTRO-AUXILIAR-2.
    WRITE DESCRIPCION-SALIDA FROM REGISTRO-AUXILIAR-2 AFTER ADVANCING
2 LINES.
PROCESO-2.
    OPEN INPUT DATOS.
    OPEN OUTPUT SALIDA-NOVEDADES.
    PERFORM PARRAFO-3.
    PERFORM PROCESO-NOVEDADES UNTIL FIN-DE-ARCHIVO EQUAL "SI".
    CLOSE DATOS.
    CLOSE SALIDA-NOVEDADES.
PROCESO-NOVEDADES.
    READ DATOS AT END MOVE "SI" TO FIN-DE-ARCHIVO.
    IF FIN-DE-ARCHIVO NOT EQUAL "SI"
        PERFORM PARRAFO-4.
*   NOELSE
*   ENDIF
PARRAFO-3.
    WRITE DESCRIPCION-SALIDA-NOV FROM LINEA-DE-BLANCOS AFTER ADVANCING
5 LINES.
    WRITE DESCRIPCION-SALIDA-NOV FROM REGISTRO-AUXILIAR-3.
PARRAFO-4.
    MULTIPLY CANTIDAD BY PRECIO-UNITARIO GIVING TOTAL-DIARIO.
    IF TOTAL-DIARIO LESS 500
        MOVE 'CAMBIAR' TO AYUDAR-CAMBIAR
        MOVE NOMBRE IN REGISTRO TO NOMBRE IN REGISTRO-AUXILIAR-4
        MOVE LOCALIZACION IN REGISTRO TO LOCALIZACION IN
REGISTRO-AUXILIAR-5
        MOVE ARTICULO IN REGISTRO TO ARTICULO IN REGISTRO-AUXILIAR-5
        WRITE DESCRIPCION-SALIDA-NOV FROM REGISTRO-AUXILIAR-4 AFTER
ADVANCING 2 LINES
        WRITE DESCRIPCION-SALIDA-NOV FROM REGISTRO-AUXILIAR-5
    ELSE
        PERFORM MAYOR-QUE-2500.
*   ENDIF

MAYOR-QUE-2500.
    IF TOTAL-DIARIO GREATER 2500
        MOVE 'AYUDAR ' TO AYUDAR-CAMBIAR
        MOVE NOMBRE IN REGISTRO TO NOMBRE IN REGISTRO-AUXILIAR-4
        MOVE LOCALIZACION IN REGISTRO TO LOCALIZACION IN
```

```
REGISTRO-AUXILIAR-5
          MOVE ARTICULO IN REGISTRO TO ARTICULO IN REGISTRO-AUXILIAR-5
          WRITE DESCRIPCION-SALIDA-NOV FROM REGISTRO-AUXILIAR-4 AFTER
ADVANCING 2 LINES
          WRITE DESCRIPCION-SALIDA-NOV FROM REGISTRO-AUXILIAR-5.
*    NOELSE
*    ENDIF
```

```
IENTIFICATION DIVISION.
ROGRAM-ID.     ORGANIZACION-RELATIVA.
AUTHOR.        SERGIO Y ADRIAN.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT ENTRADA ASSIGN TO "ENTRADA.DAT".
    SELECT RELATIVO ASSIGN TO "RELATIVO.DAT"
        ORGANIZATION IS RELATIVE.
DATA DIVISION.
FILE SECTION.
FD ENTRADA
    RECORD CONTAINS 40 CHARACTERS
    DATA RECORD IS REG-ENT.
01 REG-ENT.
    03 CARRERA-E                          PIC 9(02).
    03 NO-CUENTA-E                        PIC 9(08).
    03 NOMBRE-E                           PIC X(30).
FD RELATIVO
    RECORD CONTAINS 40 CHARACTERS
    DATA RECORD IS REG-REL
    ACCESS MODE IS RANDOM RELATIVE KEY IS LLAVE.
01 REG-REL.
    03 CARRERA-R                          PIC 9(02).
    03 NO-CUENTA-R                        PIC 9(08).
    03 NOMBRE-R                           PIC X(30).
*
WORKING-STORAGE SECTION.
77 LLAVE                                  PIC 9(02) VALUE ZERO.
77 RESP                                   PIC X(01).
*
PROCEDURE DIVISION.
INICIO.
    PERFORM ABRE-ARCHIVOS.
    READ ENTRADA AT END
        DISPLAY "ERROR, NO HAY DATOS"
        STOP RUN
    END-READ.
    PERFORM CREA-RELATIVO UNTIL CARRERA-E = 99
    DISPLAY "CON QUE REGISTRO QUIERES TRABAJAR?? (1-10),(99=FIN)"
    ACCEPT LLAVE
    DISPLAY "QUIERES MODIFICARLO?? (S/N)"
    ACCEPT RESP
    PERFORM TRABAJA-RELATIVO UNTIL LLAVE = 99
    PERFORM CIERRA-ARCHIVOS
    STOP RUN.
ABRE-ARCHIVOS.
    OPEN
        INPUT ENTRADA
        I-O   RELATIVO.
CREA-RELATIVO.
    ADD 1 TO LLAVE
```

```
        REWRITE REG-REL FROM REG-ENT INVALID KEY
            DISPLAY "ERROR EN LLAVE AL CREAR RELATIVO"
            DISPLAY "REG-REL = " REG-REL
        END-REWRITE.
        READ ENTRADA AT END
            DISPLAY "FIN DATOS DE ENTRADA, SE CREO EL RELATIVO"
        END-READ.
    TRABAJA-RELATIVO.
        IF (RESP = "S") THEN
            PERFORM MODIFICA-REGISTRO
        ELSE
            PERFORM BORRA-REGISTRO
        END-IF
        DISPLAY "CON QUE REGISTRO QUIERES TRABAJAR?? (1-10),(99-FIN)"
        ACCEPT LLAVE
        DISPLAY "QUIERES MODIFICARLO?? (S/N)"
        ACCEPT RESP.
    MODIFICA-REGISTRO.
            READ RELATIVO
                INVALID KEY
                    DISPLAY "ERROR AL LEER DEL RELATIVO"
            END-READ
            DISPLAY "REGISTRO A MODIFICAR :"
            DISPLAY REG-REL
            DISPLAY "DAME EL NUEVO REGISTRO :"
            ACCEPT REG-REL
            REWRITE REG-REL INVALID KEY
                DISPLAY "ERROR EN LLAVE AL CREAR RELATIVO"
                DISPLAY "REG-REL = " REG-REL
            END-REWRITE.
    BORRA-REGISTRO.
        READ RELATIVO
            INVALID KEY
                DISPLAY "ERROR AL LEER DEL RELATIVO"
        END-READ
        DELETE RELATIVO RECORD
            INVALID KEY
                DISPLAY "ERROR AL BORRAR EL REGISTRO"
        END-DELETE.
    CIERRA-ARCHIVOS.
        CLOSE ENTRADA
              RELATIVO.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRUEBA.
AUTHOR. GRUPO13.
INSTALLATION. VAX 11-780.
DATE-WRITTEN. 29 OCTUBRE 1984.
DATE-COMPILED. 29 OCTUBRE 1984.
SECURITY. NINGUNA.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX-11-780.
OBJECT-COMPUTER. VAX-11-780.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATOS ASSIGN "NOMBRE.DAT"
    ORGANIZATION IS SEQUENTIAL.
    SELECT NOMYRFC ASSIGN "NOMYRFC.DAT"
    ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD DATOS
    RECORD CONTAINS 36 CHARACTERS
    DATA RECORD IS REGISTRO.
01 REGISTRO.
    03 NOMBRE PIC X(30).
    03 DIA PIC 99.
    03 MES PIC 99.
    03 ANIO PIC 99.
FD NOMYRFC
    RECORD CONTAINS 40 CHARACTERS
    DATA RECORD IS SALIDA.
01 SALIDA.
    03 NOMBRE.
        05 ACOMODADO OCCURS 30 TIMES PIC X(01).
    03 LETRAS PIC A(04).
    03 ANIO PIC 99.
    03 MES PIC 99.
    03 DIA PIC 99.
WORKING-STORAGE SECTION.
77 BLANCOS PIC 99 VALUE ZEROES.
77 PUNTO-Y-COMA-1 PIC 99 VALUE ZERO.
77 PUNTO-Y-COMA-2 PIC 99 VALUE ZERO.
77 ENCONTRAR PIC X(02) VALUE "NO".
    88 YA-LA-ENCONTRE VALUE "SI".
77 I PIC 99 VALUE ZEROES.
77 J PIC 99 VALUE ZEROES.
77 LECTURA PIC X(02) VALUE "NO".
    88 FIN-DE-ARCHIVO VALUE "SI".
77 VOCALES PIC A VALUE "B".
    88 VOCAL VALUE "A","E","I","O","U".
01 NOMBRES.
    03 INDICE OCCURS 30 TIMES PIC X(01).
01 LETRAS-RFC.
```

```
    05 INDICE-LETRA OCCURS 4 TIMES PIC A(01).
PROCEDURE DIVISION.
PARRAFO-1.
PRINCIPAL.
    OPEN INPUT DATOS.
    OPEN OUTPUT NOMYRFC.
    PERFORM PROCESO UNTIL FIN-DE-ARCHIVO.
    CLOSE DATOS.
    CLOSE NOMYRFC.
    STOP RUN.
PROCESO.
    READ DATOS AT END MOVE "SI" TO LECTURA.
    MOVE ZEROES TO PUNTO-Y-COMA-1.
    MOVE ZEROES TO PUNTO-Y-COMA-2.
    MOVE ZEROES TO BLANCOS.
    MOVE ZEROES TO J.
    MOVE 1 TO I.
    MOVE "NO" TO ENCONTRAR
    IF NOT FIN-DE-ARCHIVO THEN
       PERFORM INSPECCIONA THRU IMPRIME
    ELSE
        NEXT SENTENCE.
*   ENDIF
INSPECCIONA.
    MOVE NOMBRE IN REGISTRO TO NOMBRES.
    MOVE INDICE(1) TO INDICE-LETRA(1).
    PERFORM ENCUENTRA-VOCAL UNTIL YA-LA-ENCONTRE.
    MOVE VOCALES TO INDICE-LETRA(2).
    INSPECT NOMBRES REPLACING ALL " " BY "X" BEFORE INITIAL ";".
    INSPECT NOMBRES TALLYING PUNTO-Y-COMA-1 FOR CHARACTERS BEFORE
INITIAL ";".
    ADD 2 TO PUNTO-Y-COMA-1.
    MOVE INDICE(PUNTO-Y-COMA-1) TO INDICE-LETRA(3).
    INSPECT NOMBRES REPLACING FIRST ";" BY "X"
    INSPECT NOMBRES REPLACING ALL " " BY "X" BEFORE INITIAL ";"
    INSPECT NOMBRES TALLYING PUNTO-Y-COMA-2 FOR CHARACTERS BEFORE
            INITIAL ";" REPLACING FIRST ";" BY "X"
    ADD 2 TO PUNTO-Y-COMA-2.
    MOVE INDICE(PUNTO-Y-COMA-2) TO INDICE-LETRA(4).
    MOVE CORR REGISTRO TO SALIDA.
    MOVE LETRAS-RFC TO LETRAS IN SALIDA
    INSPECT NOMBRES TALLYING BLANCOS FOR CHARACTERS BEFORE INITIAL "
".
    ADD 2 TO BLANCOS
    IF BLANCOS >30 THEN
        MOVE 30 TO BLANCOS
    ELSE
        NEXT SENTENCE.
*   ENDIF
    PERFORM ACOMODA-NOMBRE VARYING I FROM PUNTO-Y-COMA-2 BY -1 UNTIL I
= BLANCOS.
    PERFORM ACOMODA-NOMBRE VARYING I FROM 1 BY 1 UNTIL I =
PUNTO-Y-COMA-2.
```

```
    INSPECT NOMBRE IN SALIDA  REPLACING ALL "%" BY " ".
IMPRIME.
    WRITE SALIDA.

ENCUENTRA-VOCAL.
    ADD 1 TO I.
    MOVE INDICE(I) TO VOCALES.
    IF VOCAL THEN
        MOVE "SI" TO ENCONTRAR
    ELSE
        NEXT SENTENCE.
*   ENDIF

ACOMODA-NOMBRE.
    ADD 1 TO J.
    MOVE INDICE(I) TO ACOMODADO(J).
```

Opciones de la

ENVIRONMENT DIVISION

3.2 INPUT-OUTPUT SECTION

The INPUT-OUTPUT Section can contain two paragraphs: FILE-CONTROL and I-O-CONTROL.

3.3 FILE-CONTROL Paragraph

Function

The FILE-CONTROL paragraph contains file-related specifications.

General Format

```
FILE-CONTROL.
{ SELECT [ OPTIONAL ] file-name

    ASSIGN TO file-spec

    [ RESERVE reserve-num [ AREA ] ]
                          [ AREAS ]

    [ [ ORGANIZATION IS ] { SEQUENTIAL } ]
                          { RELATIVE   }
                          { INDEXED    }

* [ BLOCK CONTAINS [ smallest-block TO ] blocksize { RECORDS    } ]
                                                   { CHARACTERS }

* [ CODE-SET IS alphabet-name ]
* [ FILE STATUS IS file-stat ]
* [ ACCESS MODE IS ] SEQUENTIAL

* [ ACCESS MODE IS ] { SEQUENTIAL [ RELATIVE KEY IS rel-key ] }
                     { { RANDOM  } RELATIVE KEY IS rel-key   }
                     { { DYNAMIC }                           }
```

(continued on next page)

```
* [ ACCESS MODE IS ]  { SEQUENTIAL }
                      { RANDOM     }
                      { DYNAMIC    }

* [ RECORD KEY IS  rec-key ]

* [ ALTERNATE RECORD KEY IS  alt-key [ WITH DUPLICATES ] ] ... } ...
  [ PADDING CHARACTER IS pad-char ]
```

*These clauses are part of the Data Division File Description entry. They can be in the SELECT clause; however, they cannot be in both the SELECT clause and the File Description entry for the same file.

file-name
is the internal name of a file. Each file-name must have a File Description entry or Sort-Merge File Description entry in the Data Division. The same file-name cannot appear more than once in the FILE-CONTROL paragraph.

Syntax Rules

1. The FILE-CONTROL paragraph must have at least one SELECT clause.
2. SELECT must be the first clause in the File-Control entry. The other clauses can follow it in any order.
3. The OPTIONAL phrase can appear only for input files.

General Rules

1. There must be an OPTIONAL phrase for input files that need not be present when the program runs.
2. The rules for the OPEN statement describe the effects of the OPTIONAL phrase.

Technical Notes

The following clauses can be in either the File Description entry or the SELECT clause, but not both for the same file:

- BLOCK CONTAINS (See Section 3.3.2, BLOCK CONTAINS Clause.)
- CODE-SET (See Section 3.3.3, CODE-SET Clause.)

- RECORD (See Section 4.3.7, RECORD Clause.)
- FILE STATUS (See Section 4.3.4, FILE STATUS Clause.)
- ACCESS MODE (See Section 4.3.1, ACCESS MODE Clause.)
- RECORD KEY (See Section 4.3.8, RECORD KEY Clause.)
- ALTERNATE RECORD KEY (See Section 4.3.2, ALTERNATE RECORD KEY Clause.)

Additional References

Section 5.29 OPEN Statement

Examples

The following examples assume that the VALUE OF ID clause is not in any associated File Description entry.

1. Sequential file. This example refers to a file with sequential organization. The word INFILE is equivalent to the nonnumeric literal "INFILE". If INFILE is not a logical name at run time, the program accesses a file named "INFILE.DAT".

```
SELECT FILE-A
  ASSIGN TO INFILE.
```

2. Indexed file. This SELECT clause specifies that the indexed file need not be present when the program opens it for input.

```
SELECT OPTIONAL FILE-A
  ASSIGN TO INFILE
  ORGANIZATION INDEXED.
```

3.3.1 ASSIGN Clause

Function

The ASSIGN clause associates a file with a partial or complete file specification.

General Format

```
ASSIGN TO file-spec
```

file-spec
is a nonnumeric literal or a COBOL word formed according to the rules for user-defined names. It represents a partial or complete file specification.

General Rules

1. If there is no VALUE OF ID clause in the File Description entry, or the clause contains no file specification, file-spec is the file specification.
2. If there is a file specification in an associated VALUE OF ID clause, file-spec is the default file specification. File specification components in the VALUE OF ID clause override those in file-spec.
3. File-spec can contain a logical name.
4. If file-spec is not a literal, the compiler:
   a. Translates hyphens in the COBOL word to underline characters
   b. Treats the word as if it were enclosed in quotation marks
5. When an OPEN statement executes, VAX-11 RMS:
   a. Removes leading and trailing spaces and tab characters from the file specification
   b. Translates lower-case letters in the file specification to upper case
   c. Performs logical name translation

Additional References

Section 4.3.9 VALUE OF ID Clause

3.3.2 BLOCK CONTAINS Clause

Function

The BLOCK CONTAINS clause specifies the size of a physical record.

General Format

```
BLOCK CONTAINS [ smallest-block TO ] blksize { RECORDS    }
                                             { CHARACTERS }
```

smallest-block
is an integer literal. It specifies the minimum physical record size.

blksize
is an integer literal. It specifies the exact or maximum physical record size.

Syntax Rule

The BLOCK CONTAINS clause can be in the file's Data Division File Description entry. However, it cannot be in both the SELECT clause and the File Description entry for the same file.

General Rules

1. The BLOCK CONTAINS clause specifies physical record size.
2. The compiler ignores smallest-block.
3. The RECORDS phrase specifies physical record size in terms of logical records.
   a. For a fixed-length-record magnetic tape file, each physical record except the last contains blksize records.
   b. For a variable-length-record magnetic tape file, the compiler computes the physical record size. It equals the size of the largest logical record, plus any overhead bytes, multiplied by blksize.
   c. For a sequential disk file, there are no unused bytes in any physical record. Records can span physical record boundaries.
   d. For a relative or indexed file, the compiler uses blksize to compute the size of the physical record. Because of overhead bytes, the size can differ from record size times blksize.

4. The CHARACTERS phrase specifies physical record size in terms of characters.

   a. For files assigned to magnetic tape, the physical record size is the maximum of: (1) **blksize** bytes and (2) the size of the largest logical record; plus any overhead bytes for variable-length records.

   b. For sequential disk files, there are no unused bytes in any physical record. Records can span physical record boundaries.

   c. For relative and indexed files, the physical record size is **blksize** bytes. **Blksize** must be at least as large as the largest logical record, plus any overhead bytes. It should be a multiple of 512.

5. If there is no BLOCK CONTAINS clause, physical record size assumes a default value.

   a. For a magnetic tape file, the physical record size is the size of the largest record plus any overhead bytes.

   b. For a sequential disk file, there are no unused bytes in any physical record. Records can span physical record boundaries.

   c. For a relative or indexed file, the physical record size is the smallest number of 512-byte physical blocks that can contain at least one record (including any overhead bytes).

6. The maximum physical record size depends on file organization and device.

   a. For a sequential file, the maximum physical record size is 65,535 bytes on a magnetic tape device and 65,024 bytes on disk.

      A compile-time informational diagnostic appears if the physical record size exceeds 65,024. However, VAX-11 COBOL programs are device-independent. Therefore, a fatal run-time error can also occur if the file is assigned to disk when the image executes.

   b. For a relative or indexed file, the maximum physical record size is 16,384.

7. For files assigned to magnetic tape, the size of physical records (in characters) must be a multiple of four. Otherwise, VAX-11 RMS rounds up the physical record size to the next multiple of four.

3.3.3 CODE-SET Clause

Function

The CODE-SET clause specifies the representation of data on external media.

General Format

CODE-SET IS alphabet-name
alphabet-name is the name of a character set defined in the SPECIAL-NAMES paragraph. It cannot be described with literals in the ALPHABET clause.

Syntax Rule

The CODE-SET clause can be in the file's Data Division File Description entry. However, it cannot be in both the SELECT clause and the File Description entry for the same file.

General Rules

1. The CODE-SET clause identifies alphabet-name as the character set used to represent the file data externally.
2. Alphabet-name specifies how to convert character codes in the file to and from native character codes.
3. Code conversion occurs during execution of an input or output operation. Conversion occurs as if the data were USAGE DISPLAY.
4. Successful OPEN statement execution establishes the character set for code conversion. The set used is the one specified by alphabet-name in the File-Control entry implied by the OPEN statement.
5. If there is no CODE-SET clause, no character conversion occurs during input-output operations. The native character set is the default.

Additional References

Section 3.1.3 SPECIAL-NAMES Paragraph

Example

In this example, the CODE-SET clause specifies that the data in INFILE is coded in an alphabet named "EB". The SPECIAL-NAMES paragraph defines EB as the EBCDIC character set.

```
SPECIAL-NAMES.
    ALPHABET EB IS EBCDIC.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE ASSIGN TO INFILE
      CODE-SET IS EB.
```

3.3.4 ORGANIZATION Clause

Function

The ORGANIZATION clause specifies a file's logical structure.

General Format

```
[ ORGANIZATION IS ] { SEQUENTIAL }
                    { RELATIVE   }
                    { INDEXED    }
```

General Rules

1. File organization is fixed when the file is created. It cannot be changed after file creation.
2. If there is no ORGANIZATION clause, the default is sequential.

3.3.5 PADDING CHARACTER Clause

Function

The PADDING CHARACTER clause specifies the character to be used to pad blocks in sequential files.

General Format

```
PADDING CHARACTER IS pad-char
```

pad-char
is a one character nonnumeric literal or the data-name of a one character data item. The data-name can be qualified.

General Rules

The PADDING CHARACTER clause is for documentation only.

3.3.6 RESERVE Clause

Function

The RESERVE clause specifies the number of input-output buffers for a file.

General Format

```
RESERVE reserve-num [AREA ]
                    [AREAS]
```

reserve-num
is an integer literal from 1 through 127. It specifies the number of input-output areas for the file.

General Rule

If there is no RESERVE clause, the number of input-output areas equals the RMS default.

Technical Note

Two VAX/VMS commands change and display the defaults for multibuffering: SET RMS_DEFAULT and SHOW RMS_DEFAULT.

Additional References

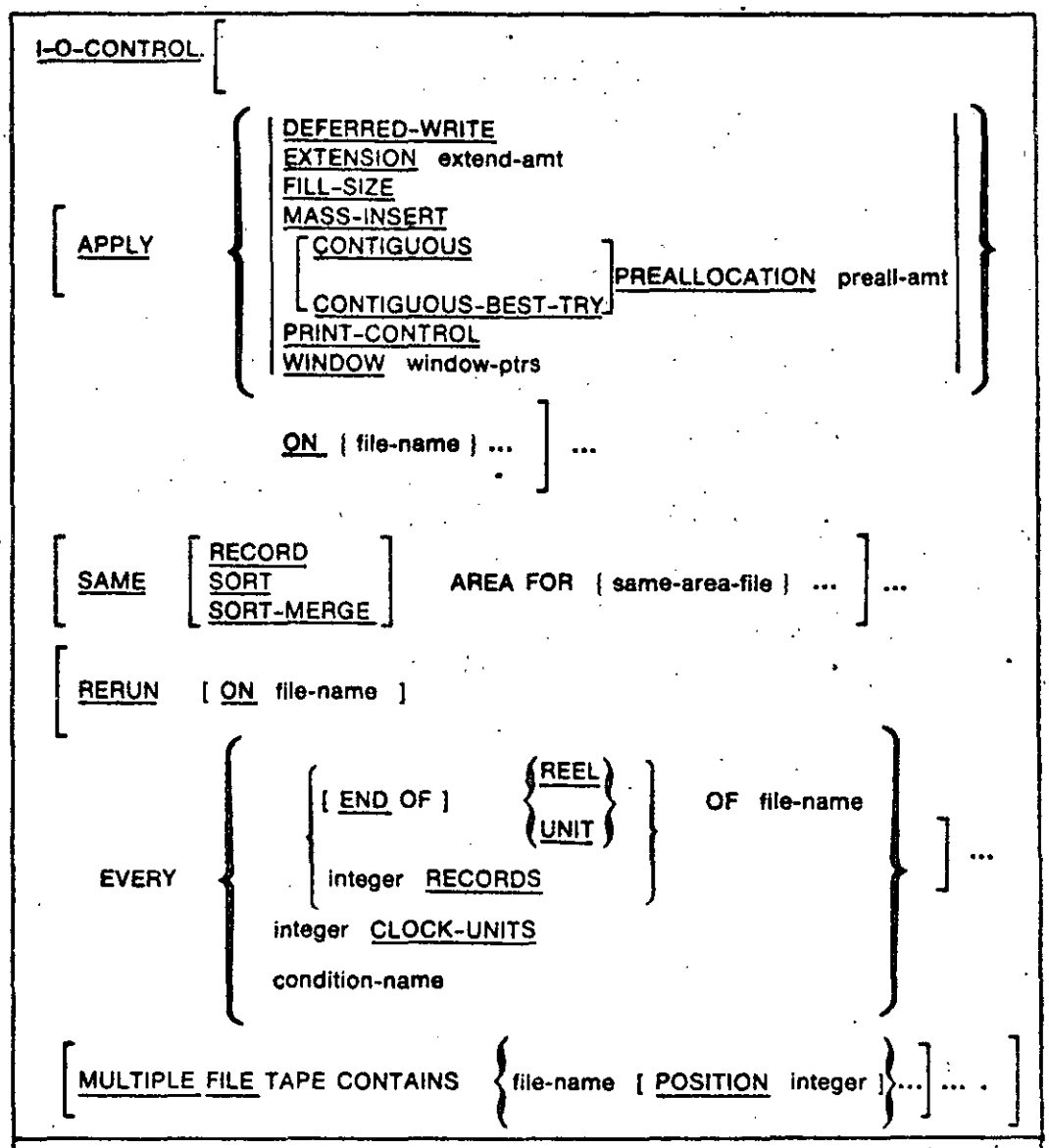
Section 3.4 APPLY Clause
VAX/VMS Command Language User's Guide

3.4 I-O-CONTROL Paragraph

Function

The I-O-CONTROL paragraph specifies the input-output techniques for a file.

General Format



extend-amt
is an integer from 0 through 65535. It specifies the number of blocks in each extension of a disk file.

(continued on next page)

prealI-amt
is an integer from 0 through 4,294,967,295. It specifies the number of blocks to allocate when the program creates a disk file.

window-ptrs
is an integer from 0 through 127. Its value can also be 255. It specifies the number of retrieval pointers in the window that maps the disk file.

file-name
is the file-name of a file described in a Data Division File Description entry.

same-area-file
is the file-name of a file to share storage areas with every other same-area-file.

Syntax Rules

1. The I-O-Control clauses can appear in any order.
2. Each APPLY clause phrase can refer to only some types of files:

Phrase	File Type
DEFERRED-WRITE	Relative or indexed organization
EXTENSION	Disk file
FILL-SIZE	Indexed organization
MASS-INSERT	Indexed organization
PREALLOCATION	Disk file
PRINT-CONTROL	Sequential organization
WINDOW	Disk file

3. More than one APPLY clause can refer to the same file-name.
4. The phrases of the APPLY clause can appear in any order. However, each phrase can be used only once for each file-name.
5. The RERUN and MULTIPLE FILE clauses cannot refer to a sort or merge file.

I-O-CONTROL (Continued)

6. In the SAME AREA clause, SORT and SORT-MERGE are equivalent.
7. If same-area-file refers to a sort or merge file, the SORT, SORT-MERGE, or RECORD phrase must be used.
8. A program can contain more than one SAME clause. However,
   a. A same-area-file cannot be in more than one SAME AREA clause.
   b. A same-area-file cannot be in more than one SAME RECORD AREA clause.
   c. A same-area-file that refers to a sort or merge file cannot be in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.
   d. If one or more same-area-files of a SAME AREA clause are in a SAME RECORD AREA clause, all same-area-files in the SAME AREA clause must be in the SAME RECORD AREA clause. However, other same-area-files can also be in the SAME RECORD AREA clause.

      The rule that only one same-area-file in a SAME AREA clause can be open at a time takes precedence over the rule that more than one same-area-file in a SAME RECORD AREA clause can be open at once.
   e. If a same-area-file that is not a sort or merge file is in a SAME AREA clause and one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, each same-area-file in the SAME AREA clause must be in the SAME SORT AREA or SAME SORT-MERGE AREA clauses.

General Rules

APPLY Clause

1. The DEFERRED-WRITE phrase causes a physical write operation to occur only when the input-output buffer for file-name is full. If there is no DEFERRED-WRITE phrase, a physical write occurs for each execution of an output statement for file-name. The DEFERRED-WRITE phrase applies only to relative and indexed files.
2. The EXTENSION phrase specifies the number of disk blocks for each extension of the file. RMS extends a file when it needs more file space to add a record.

   If extend-amt equals zero, RMS extends the file by its default value.
3. The FILL-SIZE phrase causes RMS to use the file creation fill size to fill the file's buckets. If there is no FILL-SIZE phrase, RMS fills buckets completely. The FILL-SIZE phrase applies only to indexed files.

4. The MASS-INSERT phrase optimizes the addition of records to an indexed file. However, the optimization occurs only if the records are in ascending order by Prime Record Key.

5. The PREALLOCATION phrase causes RMS to allocate preall-amt disk blocks when it creates the file.

   a. The CONTIGUOUS phrase specifies that the preallocated disk blocks must be contiguous. If RMS cannot find preall-amt contiguous disk blocks, the open fails.

   b. The CONTIGUOUS-BEST-TRY phrase causes RMS to try to preallocate disk blocks contiguously. If RMS cannot find preall-amt contiguous disk blocks, it preallocates disk blocks in the largest possible contiguous areas.

6. The PRINT-CONTROL phrase specifies that the file has print file format. The PRINT-CONTROL phrase is redundant if: (1) the File Description entry contains a LINAGE clause or (2) the program contains a WRITE statement with the ADVANCING phrase for the file. The PRINT-CONTROL phrase applies only to sequential files.

7. The WINDOW phrase causes RMS to use window-ptrs number of retrieval pointers in mapping the files. Window-ptrs must fall in the range of 0 to 127 inclusive or be equal to 255. If window-ptrs is 255, then RMS attempts to map the entire file.

SAME AREA Clause

8. The SAME AREA clause is for documentation only.

SAME RECORD AREA Clause

9. The SAME RECORD AREA clause causes two or more files named by same-area-file to share the same memory area for the current logical records.

10. More than one same-area-file (or all of them) can be open at the same time.

11. A logical record in the shared area is a logical record of:

    a. Each same-area-file of the SAME RECORD AREA clause open in the output mode

    b. The most recently read same-area-file of the SAME RECORD AREA clause open in the input mode

15

The logical records start with the same leftmost character position. Thus, the SAME RECORD AREA clause is equivalent to an implicit redefinition of the shared area.

SAME SORT (SORT-MERGE) AREA Clause

In these rules, the terms SORT, sort, and sort file imply SORT-MERGE, merge, and merge file.

12. At least one same-area-file in the SAME SORT AREA clause must be a sort file.

13. The SAME SORT AREA clause causes two or more sort files named by same-area-file to use the same memory area.

14. Files other than sort files do not share the same storage area unless their names are in a SAME AREA or SAME RECORD AREA clause.

15. No other same-area-file can be open during the execution of a SORT statement that refers to any same-area-file.

RERUN Clause

16. The RERUN clause is for documentation only. It has no effect on program execution.

MULTIPLE FILE Clause

17. The MULTIPLE FILE clause is for documentation only. It has no effect on program execution.

Technical Notes

The following notes describe the effects of APPLY clause phrases on parameters in the File Access Block (FAB) and Record Access Block (RAB) associated with file-name. Descriptions of FAB and RAB fields are in the *VAX-11 Record Management Services Reference Manual.*

1. The DEFERRED-WRITE phrase sets the DFW bit in the FOP field of the FAB.

2. The EXTENSION phrase stores extend-amt in the DEQ field of the FAB.

3. The FILL-SIZE phrase sets the LOA bit in the ROP field of the RAB.

4. The MASS-INSERT phrase sets the MAS bit in the ROP field of the RAB.

5. The PREALLOCATION phrase stores preall-amt in the ALQ field of the FAB.

   a. The CONTIGUOUS phrase sets the CTG bit in the FOP field of the FAB.

b. The CONTIGUOUS-BEST-TRY phrase sets the CBT bit in the FOP field of the FAB.

6. The PRINT-CONTROL phrase sets bits in two FAB fields:

   a. The PRN bit in the RAT field

   b. The VFC bit in the RFM field

7. The WINDOW phrase stores window-ptrs in the RTV field of the FAB.

Additional References

Section 3.3.6 RESERVE Clause

Opciones de la

DATA DIVISION

Notes:

1. This example is the same as Example 2 except that it omits ITEM-G.
2. ITEM-D is four bytes long. No fill bytes are added, since the next occurrence is already aligned on a two-byte boundary.
3. ITEM-C is 12 bytes long.
4. The record ITEM-A is 15 bytes long.

4.3 FD – File Description – Complete Entry Skeleton

Function

The File Description describes a file's physical structure, identification, and record names.

General Format

Format 1

```
FD    file-name

*     [BLOCK CONTAINS [ smallest-block TO ] blocksize {RECORDS / CHARACTERS}]

*     [ CODE-SET IS alphabet-name ]

      [RECORD {CONTAINS [ shortest-rec TO ] longest-rec CHARACTERS
               IS VARYING IN SIZE [ [ FROM shortest-rec ] [ TO longest-rec ] CHARACTERS ]
                  [ DEPENDING ON depending-item ]}]

      [LABEL [RECORDS ARE / RECORD IS] {STANDARD / OMITTED}]

      [ VALUE OF ID IS file-spec ]

      [DATA [RECORDS ARE / RECORD IS] { rec-name } ...]

      [ FILE STATUS IS file-stat ]

      [ [ ACCESS MODE IS ] SEQUENTIAL ]

      [ LINAGE IS { page-size } LINES [ WITH FOOTING AT footing-line ]
            [ LINES AT TOP top-lines ] [ LINES AT BOTTOM bottom-lines ] ]
```

Format 2

```
FD    file-name

*     [ BLOCK CONTAINS [ smallest-block TO ] blocksize {RECORDS / CHARACTERS} ]

*     [ CODE-SET IS alphabet-name ]
```

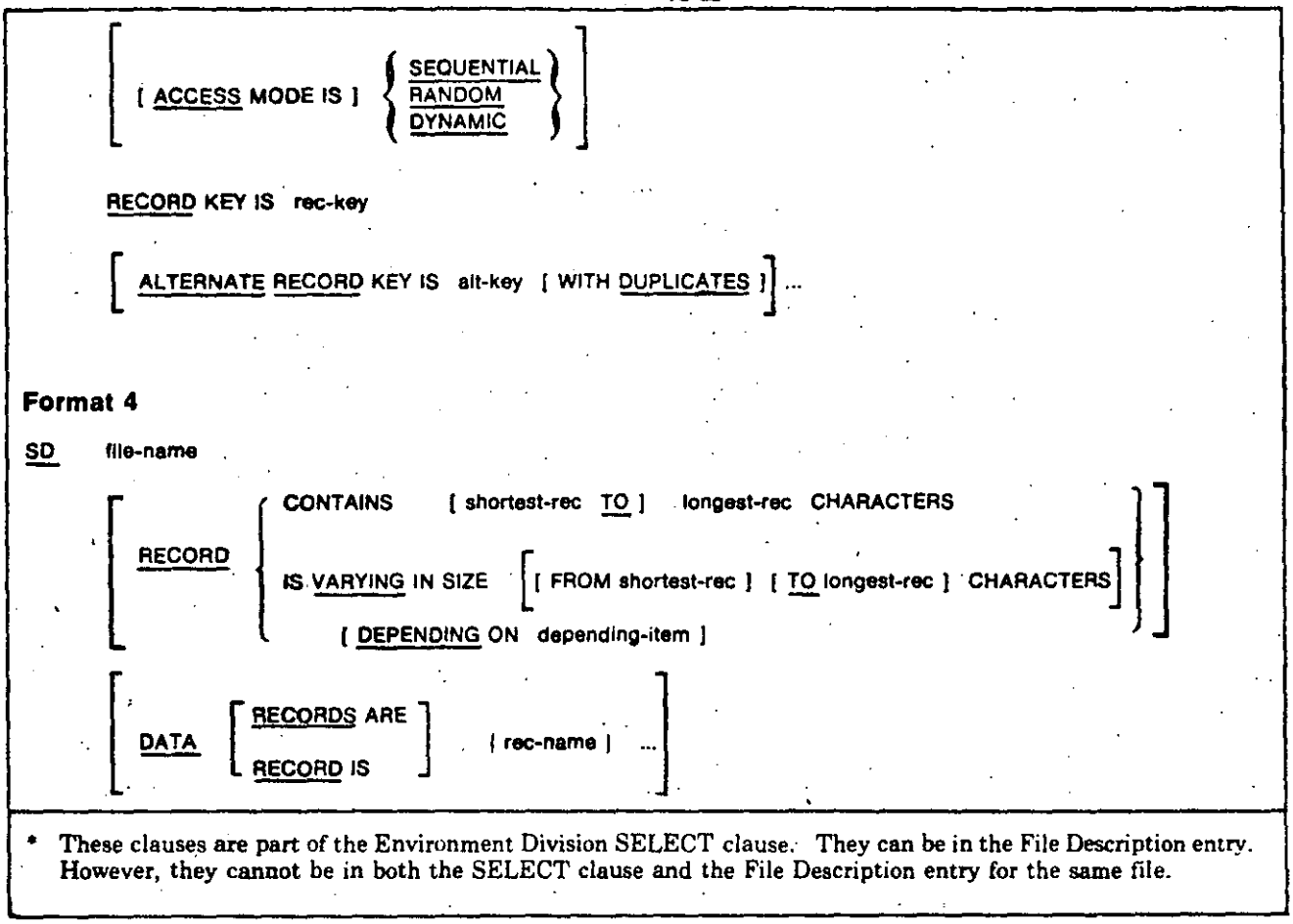
(continued on next page)

```
[ RECORD { CONTAINS [ shortest-rec TO ] longest-rec CHARACTERS
         { IS VARYING IN SIZE [ [ FROM shortest-rec ] [ TO longest-rec ] CHARACTERS ]
           [ DEPENDING ON depending-item ] } ]

[ LABEL [ RECORDS ARE ] { STANDARD } ]
        [ RECORD IS   ] { OMITTED  }

[ VALUE OF ID IS file-spec ]

[ DATA [ RECORDS ARE ] { rec-name } ... ]
       [ RECORD IS   ]

[ FILE STATUS IS file-stat ]

[ [ ACCESS MODE IS ] { SEQUENTIAL [ RELATIVE KEY IS rel-key ] } ]
                     { { RANDOM  }                            }
                     { { DYNAMIC }  RELATIVE KEY IS rel-key   } ]
```

Format 3

```
FD file-name

[ BLOCK CONTAINS [ smallest-block TO ] blocksize { RECORDS    } ]
                                                 { CHARACTERS }

[ CODE-SET IS alphabet-name ]

[ RECORD { CONTAINS [ shortest-rec TO ] longest-rec CHARACTERS
         { IS VARYING IN SIZE [ [ FROM shortest-rec ] [ TO longest-rec ] CHARACTERS ]
           [ DEPENDING ON depending-item ] } ]

[ LABEL [ RECORDS ARE ] { STANDARD } ]
        [ RECORD IS   ] { OMITTED  }

[ VALUE OF ID IS file-spec ]

[ DATA [ RECORDS ARE ] { rec-name } ... ]
       [ RECORD IS   ]

[ FILE STATUS IS file-stat ]
```

(continued on next page)



* These clauses are part of the Environment Division SELECT clause. They can be in the File Description entry. However, they cannot be in both the SELECT clause and the File Description entry for the same file.

Syntax Rules

Formats 1, 2 and 3

1. The level indicator FD identifies the start of a File Description. It must precede file-name.
2. The clauses following file-name can appear in any order.
3. A separator period must terminate a File Description entry.
4. One or more Record Description entries must follow the File Description entry.

Format 1

5. File-name can refer only to a sequential file.

Format 2

6. File-name can refer only to a relative file.
7. If a START statement refers to file-name, the File Description must have a RELATIVE KEY clause.

Format 3

8. File-name can refer only to an indexed file.

9. Alt-key cannot have the same leftmost character position as that of rec-key or any other alt-key for the same file.

Format 4

10. The level indicator SD identifies the start of a Sort-Merge File Description. It must precede file-name.

11. The clauses following file-name can appear in any order.

12. A separator period must terminate a Sort-Merge File Description entry.

13. One or more Record Description entries must follow the Sort-Merge File Description entry.

General Rules

Formats 1, 2 and 3

1. A File Description entry associates the file-name with a file connector.

Format 4

2. No input-output statements can refer to a file-name in a Sort-Merge File Description.

Technical Notes

The following clauses can be in either the File Description entry or the SELECT clause, but not both for the same file:

1. BLOCK CONTAINS (See Section 3.3.2, BLOCK CONTAINS Clause.)
2. CODE-SET (See Section 3.3.3, CODE-SET Clause.)
3. RECORD (See Section 4.3.7, RECORD Clause.)
4. FILE STATUS (See Section 4.3.4, FILE STATUS Clause.)
5. ACCESS MODE (See Section 4.3.1, ACCESS MODE Clause.)
6. RECORD KEY (See Section 4.3.8, RECORD KEY Clause.)
7. ALTERNATE RECORD KEY (See Section 4.3.2, ALTERNATE RECORD KEY Clause.)

Examples

The *VAX-11 COBOL User's Guide* contains examples of each File Description entry format.

4.3.1 ACCESS MODE Clause

Function

The ACCESS MODE clause specifies the order of access for a file's records.

General Format

Format 1

```
[ ACCESS MODE IS ]  SEQUENTIAL
```

Format 2

```
[ ACCESS MODE IS ] { SEQUENTIAL   [ RELATIVE KEY IS rel-key ] }
                   { {RANDOM  }                               }
                   { {DYNAMIC }   RELATIVE KEY IS rel-key     }
```

Format 3

```
[ ACCESS MODE IS ] { SEQUENTIAL }
                   { RANDOM     }
                   { DYNAMIC    } .
```

rel-key
is the file's RELATIVE KEY data item. It must be the data-name of an unsigned integer data item. It can be qualified but cannot be in a Record Description entry for the same file-name.

Syntax Rules

1. The ACCESS MODE clause can be in the file's SELECT clause. However, it cannot be in both the SELECT clause and File Description entry.
2. If the USING or GIVING phrases of a SORT or MERGE statement contain the name of the file, the ACCESS MODE RANDOM clause cannot be used for the file.

General Rules

All Formats

1. If there is no ACCESS MODE clause, the access mode is sequential.
2. For sequential access, record access sequence depends on file organization:
   a. Sequential files — The sequence is the same as that established by the execution of WRITE statements that created or extended the file.

24

b. Relative files — The sequence is the order of ascending relative record numbers of the file's existing records.

c. Indexed files — The sequence is the order of ascending record key values in the established Key of Reference.

Formats 2 and 3

3. For random access, the value of rel-key (for relative files) or a Record Key data item (for indexed files) indicates the record to be accessed.

4. For dynamic access, the program can access records sequentially and randomly.

Format 2

5. Relative record numbers uniquely identify records in relative files. A record's relative record number identifies its ordinal position in the file.

6. The first record in the file has a relative record number of 1. Subsequent records have consecutively higher relative record numbers.

7. The Relative Key data item associated with the execution of an input-output statement is rel-key in the File Description entry (or SELECT clause) associated with the statement.

4.3.2 ALTERNATE RECORD KEY Clause

Function

The ALTERNATE RECORD KEY clause specifies an alternate access path to indexed file records.

General Format

```
ALTERNATE RECORD KEY IS alt-key [ WITH DUPLICATES ]
```

alt-key
is the Alternate Record Key for the file. It is the data-name of a data item in a Record Description entry for the file. It can be qualified. The data item must be described as: (1) alphanumeric or alphabetic category, (2) a group item, (3) unsigned numeric display, (4) a COMP-3 integer, or (5) a COMP integer with no more than nine digits.

Syntax Rules

1. The ALTERNATE RECORD KEY clause can be in the file's SELECT clause. However, it cannot be in both the SELECT clause and File Description entry for the same file.
2. Alt-key cannot be a group item that contains a variable-occurrence data item.
3. Alt-key cannot have the same leftmost character position as that of the Prime Record Key data item or any other alt-key for the same file.

General Rules

1. The data description of alt-key and its relative location in the record must be the same as when the file was created.
2. The DUPLICATES phrase specifies that two or more records in the file can have duplicate values in the same alt-key data item. If there is no DUPLICATES phrase, two records cannot have the same value in corresponding Alternate Record Keys.
3. Only one Record Description entry for a file must describe alt-key. The Alternate Record Key has the same character positions in every record of the file.
4. A file can have up to 254 Alternate Record Keys.

4.3.3 DATA RECORDS Clause

Function

The DATA RECORDS clause documents the names of a file's Record Description entries.

General Format

```
DATA  [ RECORD IS  ]  { rec-name } ...
      [ RECORDS ARE ]
```

rec-name
is the name of a data record. It must be defined by a level 01 Record Description entry subordinate to the File Description entry.

Syntax Rule

The order of appearance of multiple rec-name entries is not significant.

General Rule

The DATA RECORDS clause is for documentation only.

4.3.4 FILE STATUS Clause

Function

The FILE STATUS clause names a data item that contains the status of an input-output operation.

General Format

FILE STATUS IS file-stat
file-stat is the data-name of a two-character alphanumeric Working-Storage Section data item. It can be qualified. File-stat is the file's FILE STATUS data item.

Syntax Rule

The FILE STATUS clause can be in the file's SELECT clause. However, it cannot be in both the SELECT clause and the File Description entry for the same file.

General Rule

After the execution of every statement that refers to the file, a value is moved to file-stat. This value indicates the statement's execution status.

Additional References

Section 5.8.9 I-O Status

4.3.5 LABEL RECORDS Clause

Function

The LABEL RECORDS clause specifies the presence or absence of labels.

General Format

General Rule

The LABEL RECORDS clause is for documentation only.

29

4.3.6 LINAGE Clause

Function

The LINAGE clause specifies the number of lines on a logical page. It can also specify the size of the logical page's top and bottom margins and the line where the footing area begins in the page body.

General Format

```
LINAGE IS { page-lines } LINES [ WITH FOOTING AT footing-line ]

  [ LINES AT TOP top-lines ] [ LINES AT BOTTOM bottom-lines ]
```

page-lines
is a positive integer or the data-name of an elementary unsigned numeric integer data item. It specifies the number of lines that can be written or spaced on the logical page. The data-name can be qualified.

footing-line
is a positive integer or the data-name of an elementary unsigned numeric integer data item. Its value cannot be greater than page-lines. Footing-line specifies the line number where the footing area begins in the page body. The data-name can be qualified.

top-lines
is an integer or the data-name of an elementary unsigned numeric integer data item. Its value can be zero. Top-lines specifies the number of lines in the top margin of the logical page. The data-name can be qualified.

bottom-lines
is an integer or the data-name of an elementary unsigned numeric integer data item. Its value can be zero. Bottom-lines specifies the number of lines in the bottom margin of the logical page. The data-name can be qualified.

General Rules

1. The LINAGE clause specifies the number of lines on a logical page.
2. Logical page size is the sum of the values specified in all phrases except FOOTING. If there is no LINES AT TOP or LINES AT BOTTOM phrase, its default value is zero. If there is no FOOTING phrase, the default value of footing-line equals the value of page-lines.

3. Logical and physical page sizes are not necessarily the same.

4. The page body is the logical page area in which the program can write or space lines. Its size equals the value of page-lines.

5. When the program opens the file by executing an OPEN statement with the OUTPUT phrase, it uses the values of page-lines, footing-line, top-lines, and bottom-lines to define the logical page sections. These values apply to all logical pages the program writes to the file during its execution.

6. The values of page-lines, top-lines, and bottom-lines affect OPEN and WRITE statement execution:

   a. When the program executes an OPEN statement with the OUTPUT phrase for the file, the values specify the number of lines in each of the associated sections of the first logical page.

   b. When the program executes a WRITE statement with the ADVANCING PAGE phrase, or when a page overflow condition occurs, the values specify the number of lines in each of the associated sections of the next logical page.

7. The value of footing-line defines the footing area for the first logical page when the program executes an OPEN statement with the OUTPUT phrase for the file. The value defines the footing area for the next logical page when: (1) the program executes a WRITE statement with the ADVANCING PAGE phrase or (2) a page overflow condition occurs.

8. The program has a special register called LINAGE-COUNTER for each file with a LINAGE clause. At any time, the value in LINAGE-COUNTER is the line number in the current page body at which the device is positioned.

9. There is a separate LINAGE-COUNTER in the program for each File Description entry that has a LINAGE clause.

10. LINAGE-COUNTER is a nine-digit numeric special register. Procedure Division statements can refer to LINAGE-COUNTER but cannot change its value.

11. If the program has more than one LINAGE-COUNTER, all Procedure Division references to it must be qualified by file-name.

12. Execution of a WRITE statement for a file with the LINAGE clause changes the value of the associated LINAGE-COUNTER:

    a. If the WRITE statement has the ADVANCING PAGE phrase, its execution resets LINAGE-COUNTER to one. The resetting implicitly increments the value of LINAGE-COUNTER to exceed the value of page-lines.

b. If the WRITE statement has the ADVANCING LINES phrase, its execution increments LINAGE-COUNTER by the value in the ADVANCING phrase.

c. If the WRITE statement does not have the ADVANCING phrase, it increments LINAGE-COUNTER by one.

13. Execution of an OPEN statement for the file sets its LINAGE-COUNTER to one.

14. Each logical page follows the one before with no spacing between them.

Technical Note

The LINAGE clause causes a file to be in print-file format. When a WRITE statement positions the file to the top of the next logical page, device positioning occurs by line spacing rather than page ejection or form feed.

The default VAX/VMS PRINT command inserts a form feed character when a form is within four lines of the bottom. Therefore, when the default PRINT command refers to a LINAGE file, unexpected page spacing can result.

The /NOFEED file qualifier of the PRINT command suppresses the insertion of form feed characters and prints LINAGE files correctly.

For example:

```
$ PRINT/NOFEED file-spec
```

Additional References

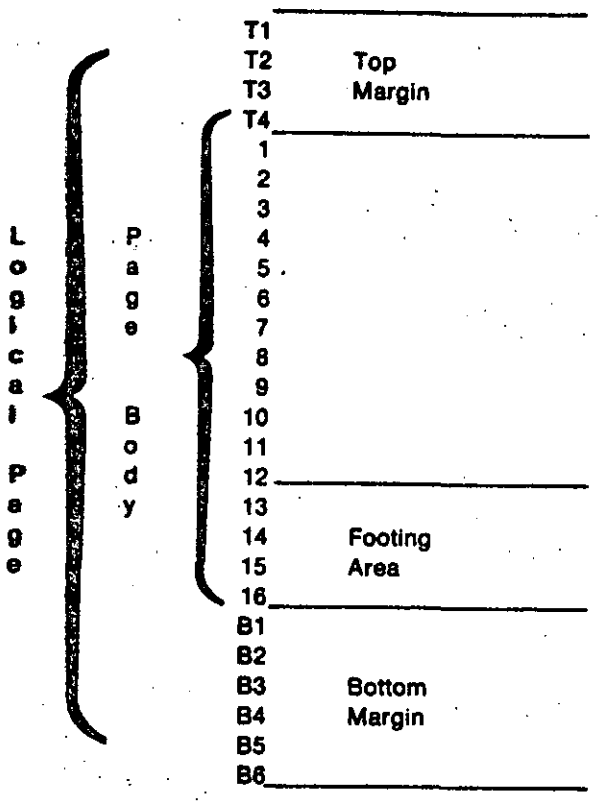
Section 5.44 WRITE Statement

Example

This example specifies a logical page whose size is 26 lines. The first line to which the page can be positioned is the fifth line. The end-of-page condition occurs when a WRITE statement causes the LINAGE-COUNTER value to be in the range 13 through 16. The page overflow condition occurs when a WRITE statement would cause the LINAGE-COUNTER value to exceed 16.

```
FD  PRINT-FILE
    VALUE OF ID IS "REPORT1.LIS"
    LINAGE IS 16 LINES WITH FOOTING AT 13
    . LINES AT TOP 4 LINES AT BOTTOM 6.
```

This figure shows the logical page areas resulting from the example:



4.3.7 RECORD Clause

Function

The RECORD clause specifies: (1) the number of character positions in a fixed-length record, (2) variable-length record format, and (3) the minimum and maximum number of character positions in a variable-length record.

General Format

Format 1

```
RECORD CONTAINS [ shortest-rec TO ] longest-rec CHARACTERS
```

Format 2

```
RECORD IS VARYING IN SIZE [ [ FROM shortest-rec ] [ TO longest-rec ] CHARACTERS ]
    [ DEPENDING ON depending-item ]
```

shortest-rec
is an integer. It specifies the minimum number of character positions in a variable-length record.

longest-rec
is an integer greater than shortest-rec. It specifies the maximum number of character positions in a variable-length record or the size of fixed-length records.

depending-item
is the data-name of an elementary unsigned integer data item in the Working-Storage or Linkage Section. It specifies the number of character positions for an output operation, and it contains the number of character positions after a successful input operation.

Syntax Rules

1. No Record Description entry for the file can specify:
   a. Fewer character positions than shortest-rec
   b. More character positions than longest-rec
2. In a Sort-Merge Description entry, the first shortest-rec character positions of the record must contain all keys specified in any SORT or MERGE statement for the sort-merge file.

3. For an indexed file, the first shortest-rec character positions of the record must contain all record keys.

General Rules

Both Formats

1. The absence of a RECORD clause is the same as a Format 1 RECORD clause with: (1) no shortest-rec phrase and (2) longest-rec equal to the greatest number of character positions described for any of the file's records.
2. The number of character positions described by a Record Description entry is the sum of:

   a. The number of character positions in all elementary items excluding redefinitions and renamings

   b. Any implicit FILLER due to alignment

   If the Record Description entry contains a table definition, the sum includes the number of character positions in the maximum number of table elements.

Format 1

3. If there is no shortest-rec phrase, Format 1 specifies fixed-length records. Longest-rec then specifies the number of character positions in each record of the file.
4. If there is a shortest-rec phrase, Format 1 specifies variable-length records, the same as Format 2 without the DEPENDING phrase.
5. For variable-length records:

   a. The maximum record size for a READ or RETURN operation is the number of character positions described in any Record Description entry for the file.

   b. During execution of a RELEASE, REWRITE, or WRITE statement, the number of character positions in a record equals the number of character positions in the Record Description entry referred to by the statement.

   c. If all Record Description entries for the file describe records of the same size, RELEASE, REWRITE, and WRITE statements for the file transfer fixed-length records in variable-length format.

Format 2

6. Format 2 specifies variable-length records.
7. If the clause does not contain shortest-rec, the minimum number of character positions in any of the file's records is the smallest number of character positions described by a Record Description entry for the file.
8. If the clause does not contain longest-rec, the maximum number of character positions in any of the file's records is the largest number of character positions described by a Record Description entry for the file.
9. If there is a DEPENDING phrase, the program must set depending-item to the number of character positions in the record before executing a RELEASE, REWRITE, or WRITE statement for the file.
10. After successfully executing a READ or RETURN statement for the file, depending-item indicates the number of character positions in the accessed record.
11. Successful DELETE and START statement executions, and unsuccessful READ and RETURN statement executions, do not change the depending-item value.
12. During RELEASE, REWRITE, and WRITE statement execution, three rules determine the number of character positions in the record:
    a. If there is a depending-item, its value specifies the number of character positions.
    b. If there is no depending-item and the record does not contain a variable-occurrence data item, the number of character positions described by the Record Description entry specifies the number of character positions.
    c. If there is no depending-item and the record contains a variable-occurrence data item, the number of character positions is the sum of the character positions in: (1) the fixed part of the record and (2) the table elements specified by the OCCURS clause depending-item when the output statement executes.

Additional References

Section 4.4.11	SYNCHRONIZED Clause
Section 4.4.12	USAGE Clause

4.3.8 RECORD KEY Clause

Function

The RECORD KEY clause specifies the Prime Record Key access path to indexed file records.

General Format

RECORD KEY IS rec-key

rec-key
is the data-name of a data item in a Record Description entry for the file. It can be qualified, but it cannot be a group item that contains a variable-occurrence data item. The data item must be described as: (1) alphanumeric or alphabetic category, (2) a group item, (3) unsigned numeric display, (4) a COMP-3 integer, or (5) a COMP integer with no more than nine digits.

Syntax Rule

The RECORD KEY clause can be in the file's SELECT clause. However, it cannot be in both the SELECT clause and the File Description entry for the same file.

General Rules

1. The RECORD KEY clause specifies the Prime Record Key for a file.
2. The values of the Prime Record Key cannot be duplicated in the file's records.
3. The data description of rec-key, and its relative location in the record, must be the same as those used when the file was created.
4. Only one Record Description entry for the file must describe rec-key. The Prime Record Key has the same character positions in every record of the file.

4.3.9 VALUE OF ID Clause

Function

The VALUE OF ID clause specifies, replaces, or completes a file specification.

General Format

VALUE OF ID IS file-spec
file-spec is a nonnumeric literal or the data-name of an alphanumeric Working-Storage Section data item. It contains the full or partial file specification. File-spec can be qualified.

Technical Notes

1. File-spec is a complete or partial file specification in RMS format.
2. Each file specification field in file-spec augments the specification in the SELECT clause ASSIGN phrase.
3. A file specification field in file-spec overrides the corresponding field in the SELECT clause. If a file specification field is in either the SELECT clause or file-spec, but not both, it becomes part of the file specification.
4. If the program opens a file in the input, I-O, or extend mode, RMS uses file-spec to locate the existing file.
5. If the program opens a file in the output mode, or the file does not exist, RMS uses file-spec to name the new file.

Additional References

VAX-11 COBOL User's Guide Input-Output Handling

Opciones de la

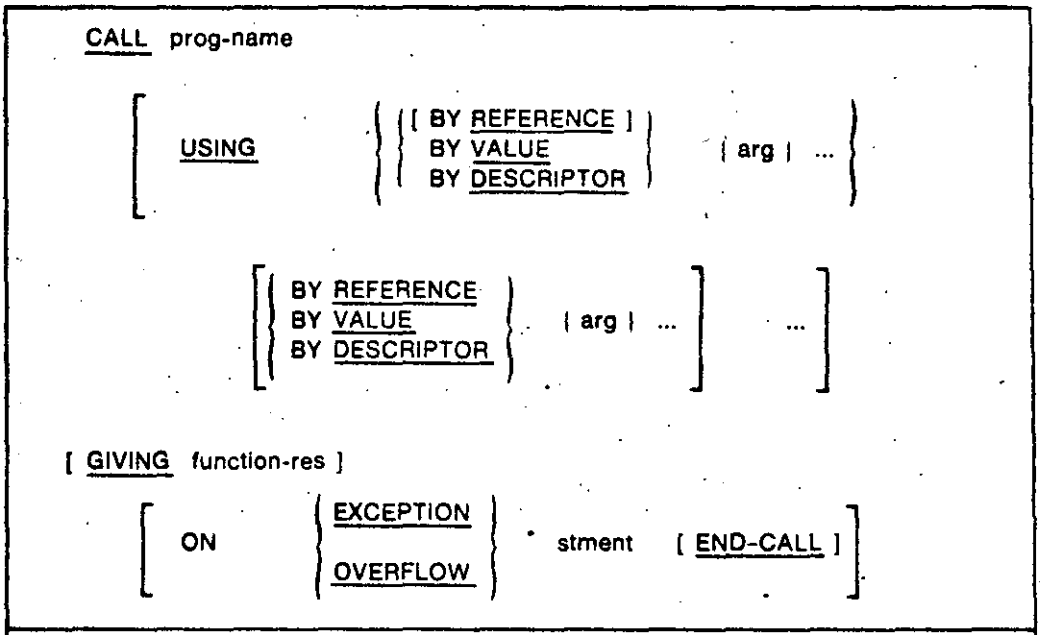
PROCEDURE DIVISION:

5.13 CALL Statement

Function

The CALL statement transfers control to another program in the executable image.

General Format



prog-name
is a nonnumeric literal or the identifier of an alphanumeric data item. It is the name of the program to which control transfers.

arg
is the argument. It identifies the data that is available to both the calling and called programs. It is the data-name of a level–01 or elementary data item described in the Data Division. Arg can be qualified when it refers to a File Section data item. If the argument-passing mechanism is BY VALUE, arg can also be: (1) a signed integer numeric literal in the range –2**31 to +2**31–1, (2) an unsigned integer numeric literal in the range 0 to 2**31–1, (3) a COMP–1 data item, or (4) a word or longword COMP.

function-res
is the identifier of an elementary integer numeric data item with COMP, COMP-1, or COMP-2 usage and no scaling positions. Function-res can be subscripted, and it can be qualified. When control returns to the calling program, function-res can contain a function result.

stment
is an imperative statement.

Syntax Rules

1. Prog-name must be from one to 30 characters long. It can contain the characters A–Z, a-z, 0–9, $ (dollar sign), - (hyphen), and _ (underline).
2. Prog-name is the entry-point in the called program. For COBOL programs, prog-name is the program-name in the PROGRAM-ID paragraph.
3. The same arg can appear more than once in the USING phrase.
4. The maximum number of arguments is 255.
5. If there is no initial argument-passing mechanism (REFERENCE, VALUE, or DESCRIPTOR), BY REFERENCE is the default.
6. An argument-passing mechanism applies to every arg following it until a new mechanism (if any) appears.
7. The CALL statement has a USING phrase only if there is a USING phrase in the Procedure Division header of the called program. Both USING phrases must have the same number of arguments.

General Rules

1. The program whose name is specified by prog-name is the called program. The program containing the CALL statement is the calling program.
2. When the CALL statement executes, the contents of prog-name are interpreted as follows:
   a. Hyphens are treated as underline characters.
   b. Lowercase letters are treated as upper case.
   c. Leading and trailing spaces and tab characters are ignored.
3. The CALL statement transfers control to the called program.
4. If prog-name is an identifier, the CALL statement can transfer control only to VAX-11 COBOL programs.
5. If prog-name (in an identifier) is not in the executable image and there is an ON EXCEPTION phrase, control does not transfer; stment executes.
6. If prog-name (in an identifier) is not in the executable image and there is no ON EXCEPTION phrase, an error condition exists; the program terminates abnormally.
7. If the called program does not have the initial attribute, it is in its initial state: (1) the first time it is called in an image, and (2) the first time it is called after a CANCEL to the called program.

   On all other entries, the state of the called program is the same as when it was last exited. The program state includes internal data.

8. If the called program has the initial attribute, it is in its initial state every time it is called.

9. Files associated with a called program's file connectors are not in the open mode:

   a. The first time the program is called

   b. The first time the program is called after execution of a CANCEL statement referring to the program

   c. Every time the program is called, if it has the initial attribute

   On all other entries, the status and positioning of files in a called program are the same as when the program was last exited.

10. The arguments' order of appearance in the USING phrases of the CALL statement and the called program's Procedure Division header determines correspondence between the data-names used by the calling and called programs. Data-names correspond by position in the USING phrase—not by name.

    No correspondence exists for index-names. Therefore, index-names in calling and called programs always refer to separate indexes.

11. The arguments in the CALL statement USING phrase are made available to the called program when the CALL executes.

12. Called programs can contain CALL statements. However, a called program must not execute a CALL statement that directly or indirectly calls the calling program.

13. The CALL statement can make data available to the called program by three argument-passing mechanisms:

    a. REFERENCE - The address of (pointer to) arg is passed to the called program. This is the default mechanism: arguments are passed by REFERENCE if there is no explicit mechanism in the CALL statement.

    b. VALUE - The value of arg is passed to the called program. If arg is a data-name, its description in the Data Division can be:

       - COMP usage with no scaling positions. The picture can specify no more than nine digits.
       - COMP-1 usage

    c. DESCRIPTOR - The address of (pointer to) the data item's descriptor is passed to the called program.

14. If the called program is a COBOL program, the CALL statement can pass arguments only BY REFERENCE. Otherwise, the mechanism for each argument in the CALL statement USING phrase must be the same as the mechanism for each argument in the non-COBOL called program's argument list.

Additional References

Chapter 5 Procedure Division Header
VAX-11 Architecture Handbook

Examples

1. Passing arguments by reference.

```
CALL "DATERTN" USING ITEMA ITEMB ITEMC.
```

2. Mixing argument-passing mechanisms. Reference arguments are ITEMA and ITEMD. Descriptor arguments are ITEMB, ITEMC, ITEMD, and ITEMF. The value argument is ITEME. ITEMD is passed twice; by reference and by descriptor.

```
CALL "NEWPROG" USING ITEMA
  BY DESCRIPTOR ITEMB ITEMC
  REFERENCE ITEMD
  VALUE ITEME
  DESCRIPTOR ITEMD ITEMF.
```

3. Calling a program whose name is selected at run time.

```
MOVE "PROG009" TO PROG-TO-CALL.
...
CALL PROG-TO-CALL USING ITEMA.
```

4. Receiving a function result.

```
CALL "PROG010" USING ITEMA ITEMB
  GIVING ITEMC.
```

43

5.17 CONTINUE Statement

Function

The CONTINUE statement indicates that no executable statement is present. It causes an implicit control transfer to the next executable statement.

General Format

```
CONTINUE
```

Syntax Rule

The CONTINUE statement can be used wherever a conditional or imperative statement can be used.

General Rule

The CONTINUE statement causes an implicit control transfer to the next executable statement.

Example

This example shows how CONTINUE can replace an INVALID KEY imperative statement. Control passes to the MOVE statement whether or not the INVALID KEY condition occurs.

```
READ FILE-A
  INVALID KEY
    CONTINUE.
MOVE ...
```

44

5.18 DELETE Statement

Function

The DELETE statement logically removes a record from a mass storage file.

General Format

```
DELETE file-name RECORD [ INVALID KEY stment [ END-DELETE ] ]
```

file-name
is the name of a file described in the Data Division. It cannot be the name of a sequential file or a sort or merge file.

stment
is an imperative statement.

Syntax Rules

1. There cannot be an INVALID KEY phrase for a DELETE statement that references a file in sequential access mode.
2. There must be an INVALID KEY phrase if: (1) the file is not in sequential access mode and (2) there is no applicable USE AFTER EXCEPTION procedure.

General Rules

1. The file must be open in I-O mode when the DELETE statement executes.
2. For a sequential access file, a successfully executed READ statement must be the last input-output statement executed for the file before the DELETE statement. RMS logically removes the record the READ statement accessed.
3. For a relative file in random or dynamic access mode, RMS logically removes the record identified by the file's RELATIVE KEY data item. If the file does not contain that record, an invalid key condition exists.
4. For an indexed file in random or dynamic access mode, RMS logically removes the record identified by the file's Prime Record Key data item. If the file does not contain that record, an invalid key condition exists.
5. After successful DELETE statement execution, the identified record has been logically removed from the file. It is no longer accessible.
6. DELETE statement execution does not affect the contents of the record area. It also does not affect the contents of the data item referred to in the DEPENDING ON phrase of the file's RECORD clause.

7. For sequential access files, DELETE statement execution does not affect the Next Record Pointer.

8. For dynamic access files, the Next Record Pointer can point to the deleted record before the DELETE. After the DELETE statement executes, the Next Record Pointer:

   a. Points to a *relative* file's next existing record

   b. Points to an *indexed* file's next existing record, as established by the Key of Reference

   c. Indicates the at end condition if the file has no next record

9. DELETE statement execution updates the value of the FILE STATUS data item for the file.

> If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a non-zero value in a FILE STATUS data item. However, it does not execute if the condition is invalid key, and there is an INVALID KEY phrase.

Technical Notes

1. DELETE statement execution can result in the following FILE STATUS data item values:

FILE STATUS	Access Method	Meaning
00	All	Successful
23	Rand	Record not in file (invalid key)
92	All	Record locked by another program
93	Seq	No previous READ or START
94	All	File not open, or incompatible open mode
30	All	All other permanent errors

Additional References

Section 5.3	Scope of Statements
Section 5.29	OPEN Statement
Section 5.8.10	Invalid Key Condition
Section 5.8.9	I-O Status

16

5.19 DISPLAY Statement

Function

The DISPLAY statement transfers low-volume data from the program to the default system output device or the object of a mnemonic-name.

General Format

```
DISPLAY  { src-item }  ...  [ UPON  output-dest ]
    [ WITH NO ADVANCING ]
```

src-item
is a literal or the identifier of a data item. If the literal is a figurative constant, it cannot be "ALL literal."

output-dest
is a mnemonic-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

Syntax Rule

In a DISPLAY statement, the number of src-item entries cannot exceed 254.

General Rules

1. The DISPLAY statement transfers data from each src-item (in its order of appearance in the statement) to output-dest.
2. No editing or conversion occurs during DISPLAY execution.
3. If src-item is a figurative constant, only one occurrence is displayed.
4. When there is more than one src-item, sending item size is the sum of the src-item sizes. The DISPLAY statement does not transfer any device-positioning information between consecutive src-item values.
5. If there is no UPON phrase, the DISPLAY statement transfers data to the default system output device. If there is a WITH NO ADVANCING phrase, the DISPLAY statement does not transfer any device positioning information after the last src-item value.
6. If there is no WITH NO ADVANCING phrase, the DISPLAY statement transfers device positioning information. It resets the output-dest position to the leftmost position on its next line.

Technical Notes

1. The DISPLAY statement transfers data through VAX-11 RMS using the Variable with Fixed-Length Control (VFC) format.
2. The default system output device for COBOL programs is the VAX/VMS logical name COB$OUTPUT. Therefore, a DISPLAY statement without the UPON phrase transfers data to the object of the logical name COB$OUTPUT.
3. When there is an UPON phrase, DISPLAY transfers data to the object of the VAX/VMS logical name associated with the SPECIAL-NAMES paragraph description of output-dest.
4. The object of a logical name is not necessarily a device. Therefore, there is no implication of open mode. As a result, output-dest can be associated with any device-name in the SPECIAL-NAMES paragraph. For example, output-dest can refer to PAPER-TAPE-READER as well as PAPER-TAPE-PUNCH.

Additional References

Section 3.1.3 SPECIAL-NAMES Paragraph

Examples

In the example results, the character "s" represents a space. The examples assume the following Environment and Data Division entries:

```
SPECIAL-NAMES.
    LINE-PRINTER IS ERR-REPORTER.

01  ITEMA PIC X(6) VALUE "ITEMS".
01  ITEMB PIC X(8) VALUE "VALID".
01  ITEMC PIC X(5) VALUE "TODAY".
01  ITEMD PIC 99 VALUE 2.
01  ITEME PIC X(10) VALUE "MONDAY".
```

```
1. DISPLAY ITEMC.                                   TODAY

2. DISPLAY ITEMD UPON ERR-REPORTER.                 02

3. DISPLAY ITEMD ITEMA "ARE" ITEMB.                 02ITEMSsAREVALIDsss

4. DISPLAY ITEMD SPACE ITEMA "AREs" ITEMB.          02sITEMSsAREsVALIDsss

5. DISPLAY ITEMC "sISs" NO ADVANCING.               TODAYsISsMONDAYssss
   DISPLAY ITEME.                                   MONDAYssss
   DISPLAY ITEME.
```

48

5.21 EXIT Statement

Function

The EXIT statement provides a common logical end point for a series of procedures. It also marks the logical end of a called program.

General Format

```
EXIT [ PROGRAM ]
```

Syntax Rules

1. The EXIT statement without the PROGRAM phrase must be the only statement in the sentence. It must also be the only sentence in the paragraph.
2. If EXIT PROGRAM is in a consecutive sequence of imperative statements in a sentence, it must be the last statement in the sentence.

General Rules

1. EXIT without the PROGRAM phrase associates a procedure-name with a point in the program. It has no other effect on program compilation or execution.
2. If EXIT PROGRAM executes in a program that is not a called program, it has the same effect as a STOP RUN statement; image execution ends.
3. If EXIT PROGRAM executes in a called program without the INITIAL clause in its PROGRAM-ID paragraph, execution continues with the next executable statement after the CALL statement in the calling program.

   The state of the calling program does not change. It is the same as when the program executed the CALL statement. However, the contents of data items shared by the calling and called programs may have been changed.

   The state of the called program does not change. However, the called program is considered to have reached the ends of the ranges of all PERFORM statements it executed.
4. When EXIT PROGRAM executes in a called program with the initial attribute, the actions described in General Rule 3 also apply. In addition, executing the EXIT PROGRAM statement is equivalent to also executing a CANCEL statement that names the called program.

Examples

1.
```
PROC-A.
    EXIT.
```

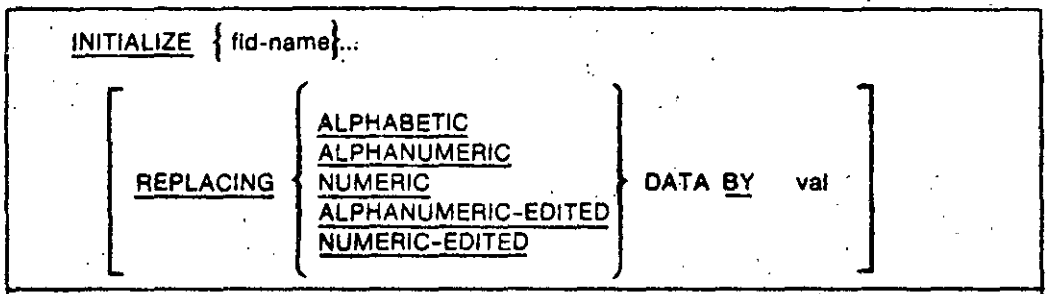
2.
```
TEST-RETURN.
    IF ITEMA NOT = ITEMB
      MOVE ITEMA TO ITEMB
      EXIT PROGRAM.
```

5.24 INITIALIZE Statement

Function

The INITIALIZE statement sets selected types of data fields to predetermined values.

General Format



fld-name
is the identifier of the receiving area data item.

val
is the sending area. It can be a literal or the identifier of a data item.

Syntax Rules

1. The phrase after the word REPLACING is the category phrase.
2. The category of the data item referred to by val must be consistent with that in the category phrase. The combination of categories must allow execution of a valid MOVE statement.
3. If a fld-name is an elementary item, its category must be the same as that in the category phrase.
4. The description of fld-name or any item subordinate to it cannot contain the OCCURS clause DEPENDING phrase.
5. Neither fld-name nor val can be index data items.

General Rules

1. Fld-name can be an elementary or group item. However, all data movement operations occur as if they resulted from a series of MOVE statements with elementary item receiving areas:
   a. If the receiving area is a group item, INITIALIZE affects only those subordinate elementary items whose category matches the category phrase. General Rule 5 describes the effect on elementary items when there is no REPLACING phrase.

b. INITIALIZE affects all eligible elementary items, including all occurrences of table items in the group.

2. INITIALIZE does not affect index data items and FILLER data items.
3. INITIALIZE does not affect items subordinate to fld-name that contain a REDEFINES clause. Nor does it affect data items subordinate to those items. However, fld-name itself can have a REDEFINES clause or be subordinate to a data item that does.
4. When there is a REPLACING phrase, val is the sending field for each of the implicit MOVE statements.
5. When there is no REPLACING phrase, the sending field for the implicit MOVE statements is:
   a. SPACES, if the data item category is alphabetic, alphanumeric, or alphanumeric edited
   b. ZEROS, if the data item category is numeric or numeric edited
6. INITIALIZE operates on each fld-name in the order it appears in the statement. When fld-name is a group item, INITIALIZE operates on its eligible subordinate elementary items in the order they are defined in the group.

Additional References

Section 5.27 MOVE Statement

Examples

In the examples' results, "–" means that the value of the data item is unchanged; "s" represents the character space. The examples assume this data description:

```
01  ITEMA.
    03  ITEMB       PIC X(4).
    03  ITEMC.
        05  ITEMD   PIC 9(5).
        05  ITEME   PIC $$$9.99.
        05  ITEMF   PIC XX/XX.
    03  ITEMG.
        05  ITEMH   PIC 999.
        05  ITEMI   PIC XX.
        05  ITEMJ   PIC 99.9.
    03  ITEMK       PIC X(4) JUSTIFIED RIGHT.
```

1. `INITIALIZE ITEMA.`
2. `INITIALIZE ITEMB ITEMG.`
3. `INITIALIZE ITEMA REPLACING ALPHANUMERIC BY "ABCDE".`
4. `INITIALIZE ITEMG REPLACING NUMERIC BY 9.`
5. `INITIALIZE ITEMA REPLACING NUMERIC-EDITED BY 16.`
6. `INITIALIZE ITEMA REPLACING ALPHANUMERIC-EDITED BY "ABCD".`
7. `INITIALIZE ITEMA REPLACING ALPHANUMERIC BY "99".`

	ITEMB	ITEMD	ITEME	ITEMF	ITEMH	ITEMI	ITEMJ	ITEMK
1.	ssss	00000	ss$0.00	ss/ss	000	ss	00.0	ssss
2.	ssss	-	-	-	000	ss	00.0	-
3.	ABCD	-	-	-	-	AB	-	BCDE
4.	-	-	-	-	009	-	-	-
5.	-	-	s$16.00	-	-	-	16.0	-
6.	-	-	-	AB/CD	-	-	-	-
7.	99ss	-	-	-	-	99	-	ss99

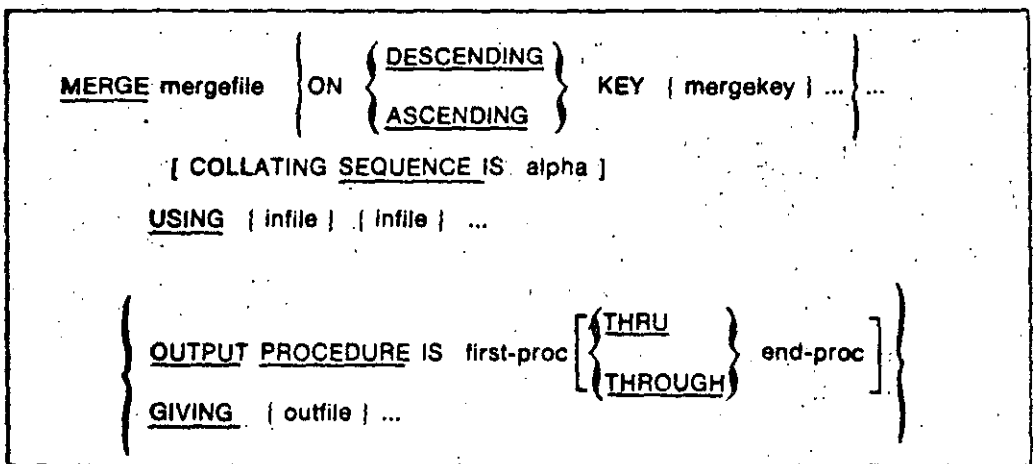
53

5.26 MERGE Statement

Function

The MERGE statement combines two or more identically sequenced files on a set of key values. During the process, it makes records available, in merged order, to an output procedure or an output file.

General Format



mergefile
is a file-name described in a Sort-Merge File Description (SD) entry in the Data Division.

mergekey
is the data-name of a data item in a record associated with mergefile.

alpha
is an alphabet-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

infile
is the file-name of the input file. It must be described in a File Description (FD) entry in the Data Division.

first-proc
is the section-name of the output procedure's first section.

end-proc
is the section-name of the output procedure's last section.

outfile
is the file-name of the output file. It must be described in a File Description (FD) entry in the Data Division.

Syntax Rules

1. MERGE statements can be anywhere in the Procedure Division except in:
   a. Declaratives
   b. A SORT or MERGE statement input or output procedure
2. If mergefile contains variable-length records, infile records must not be smaller than the smallest in mergefile nor larger than the largest.
3. If mergefile contains fixed-length records, infile records must not be larger than the largest record described for mergefile.
4. If outfile contains variable-length records, mergefile records must not be smaller than the smallest in outfile nor larger than the largest.
5. If outfile contains fixed-length records, mergefile records must not be larger than the largest record described for outfile.
6. Each mergekey must be described in records associated with mergefile.
7. Mergekey can be qualified.
8. Mergekey cannot be a group that contains variable-occurrence data items.
9. The description of mergekey cannot contain an OCCURS clause or be subordinate to one that does.
10. Mergefile can have more than one record description. However, mergekey need not be described in more than one of the record descriptions. The character positions referenced by mergekey are used as the key for all the file's records.
11. The words THRU and THROUGH are equivalent.
12. If outfile is an indexed file, the first mergekey must be in the ASCENDING phrase. It must specify the same character positions in its record as the Prime Record Key for outfile.

General Rules

1. The MERGE statement merges all records in the infile files.
2. If mergefile contains fixed-length records, any shorter infile records are space-filled on the right after the last character. Space-filling occurs before the infile record is released to mergefile.
3. The leftmost mergekey is the major key, and the next mergekey is the next most significant key. The significance of mergekey data items is not affected by how they are divided into KEY phrases. Only left-to-right order determines significance.

4. The ASCENDING phrase causes the merged sequence to be from the lowest mergekey value to the highest.
5. The DESCENDING phrase causes the merged sequence to be from the highest mergekey value to the lowest.
6. Merge sequence follows the rules for relation condition comparisons.
7. When the contents of all key data items of one record equal the contents of the corresponding key data items in another record, the order of return from the merge:
   a. Follows the order of the associated input files in the MERGE statement
   b. Causes all records with equal key values from one input file to be returned before any are returned from another.
8. The MERGE statement determines the comparison collating sequence for nonnumeric mergekey items when it begins execution. If there is a COLLATING SEQUENCE phrase in the MERGE statement, MERGE uses that sequence. Otherwise, it uses the program collating sequence.
9. The results of the merge are undefined unless the records in the infile files are ordered as described in the MERGE statement's ASCENDING or DESCENDING KEY clause.
10. The MERGE statement transfers all records in infile to mergefile. When the MERGE statement executes, infile must not be open.
11. For each infile, the MERGE statement:
    a. Begins file processing as if the program had executed an OPEN statement with the INPUT phrase
    b. Gets the logical records and releases them to the merge operation. MERGE obtains each record as if the program had executed a READ statement with the NEXT and AT END phrases.
    c. Terminates file processing as if the program had executed a CLOSE statement with no optional phrases.

    These implicit OPEN, READ, and CLOSE operations can cause associated USE procedures to execute.
12. The output procedure consists of one or more sections that:
    - Are contiguous in the source program
    - Do not form a part of any other procedure

13. When the MERGE statement enters the output procedure, it is ready to select the next record in merged order. The output procedure must execute at least one RETURN statement to make records available for processing.

14. The program must not pass control to the output procedure except during execution of a related MERGE statement.

15. The output procedure cannot contain SORT or MERGE statements. It must not explicitly transfer control outside the output procedure. However, statements can cause implied control transfers to Declaratives.

16. The remainder of the Procedure Division must not transfer control to points in the output procedure.

17. If the MERGE statement is in a *fixed* segment, the output procedure must be either:

    a. Completely in fixed segments

    b. Completely contained in one independent segment

18. If the MERGE statement is in an *independent* segment, the output procedure must be either:

    a. Completely in fixed segments

    b. Completely contained in the same independent segment as the MERGE statement itself

19. If there is an output procedure, control passes to it during execution of the MERGE statement. When control passes the last statement in the output procedure's last section, the MERGE statement ends. Control transfers to the next executable statement after the MERGE statement.

20. During execution of the output procedures, or USE AFTER EXCEPTION procedure implicitly invoked during the MERGE statement, no statement can manipulate the files or record areas associated with infile or outfile.

21. If there is a GIVING phrase, the MERGE statement writes all merged records to each outfile. This transfer is an implied MERGE statement output procedure. When the MERGE statement executes, outfile must not be open.

22. The MERGE statement begins outfile processing as if the program had executed an OPEN statement with the OUTPUT phrase.

23. The MERGE statement gets the merged logical records and writes them to each outfile. MERGE writes each record as if the program had executed a WRITE statement with no optional phrases.

For relative files, the value of the relative key data item is 1 for the first returned record, 2 for the second, and so on. When the MERGE statement ends, the value of the Relative Key data item indicates the number of outfile records.

24. The MERGE statement terminates outfile processing as if the program had executed a CLOSE statement with no optional phrases.

25. These implicit OPEN, WRITE and CLOSE operations can cause associated USE procedures to execute. If the MERGE statement tries to write beyond the boundaries of outfile, the applicable USE AFTER EXCEPTION procedure executes. If control returns from the USE procedure, or if there is none, outfile processing terminates as if the program had executed a CLOSE statement with no optional phrases.

26. If outfile contains fixed-length records, any shorter mergefile records are space-filled on the right after the last character. Space-filling occurs before the mergefile record is released to outfile.

Additional References

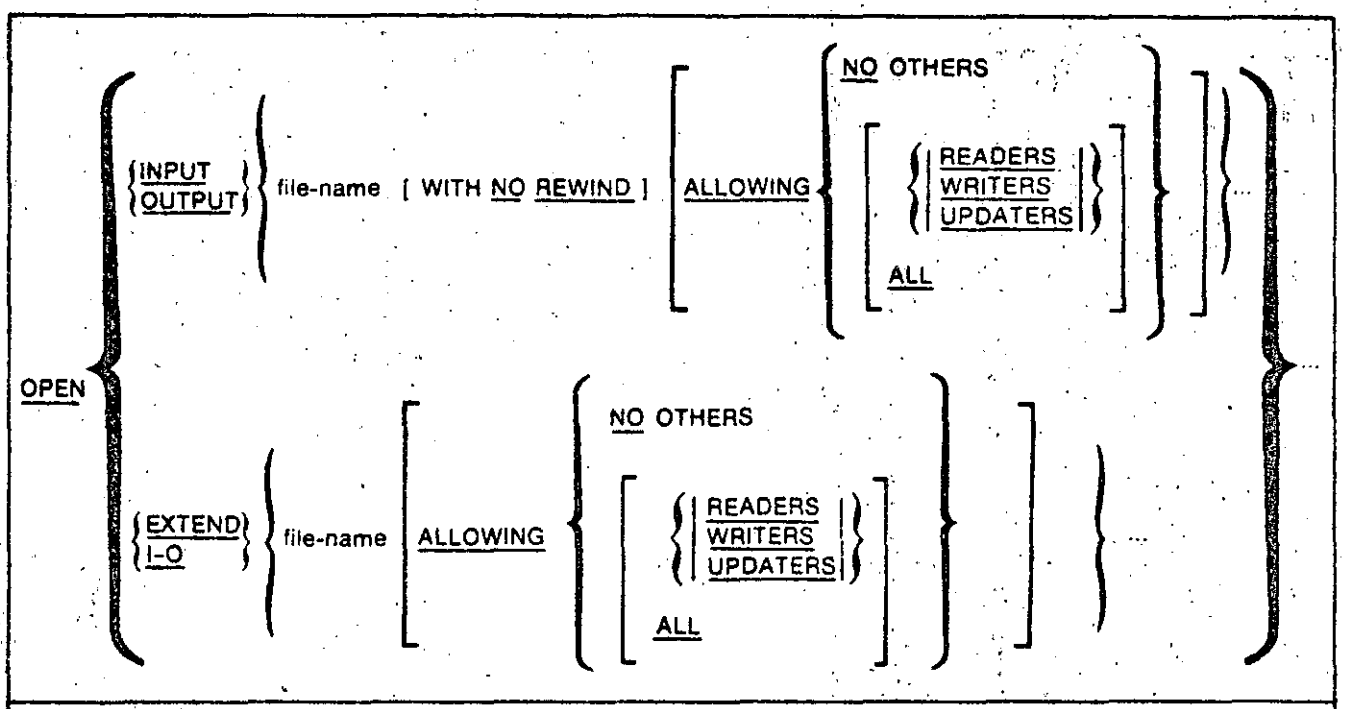
Section 3.1.2	OBJECT-COMPUTER Paragraph
Section 3.1.3	SPECIAL-NAMES Paragraph
Section 3.4	I-O-CONTROL Paragraph
VAX-11 COBOL User's Guide	Sorting and Merging

5.29 OPEN Statement

Function

The OPEN statement begins the processing of a file and makes it available to the program.

General Format



file-name
is the name of a file described in the Data Division. It cannot be the name of a sort or merge file.

Syntax Rules

1. The NO REWIND phrase can be used only for sequential files.
2. The I-O phrase can be used only for mass storage files.
3. The EXTEND phrase can be used only for sequential access mode files.

General Rules

1. Successful OPEN statement execution:
   a. Makes the file available to the program
   b. Puts the file in an open mode
   c. Associates the file with the file-name through the file connector

2. A file is available if it is both:

   a. Physically present

   b. Recognized by RMS

   Table 5-6 shows the result of opening available and unavailable files.

Table 5-6: Opening Available and Unavailable Files

Open Mode	File Is Available	File Is Unavailable
INPUT	Normal open	Error
INPUT (Optional File)	Normal open	Normal open. The first read causes the at end condition or invalid key condition
I-O (RANDOM or DYNAMIC access)	Normal open	The open creates the file.
I-O (SEQUENTIAL access)	Normal open	Error
OUTPUT	Creates a new version of the file.	The open creates the file
EXTEND	Normal open	The open creates the file.

3. Successful OPEN statement execution makes the file's record area available to the program.

4. When a file is not in an open mode, no statement that references the file either implicitly or explicitly can execute, except for:

   a. A MERGE statement with the GIVING phrase

   b. An OPEN statement

   c. A SORT statement with the USING or GIVING phrase

5. An OPEN statement for a file must successfully execute before any allowable input-output statement executes for the file. Table 5-7 shows allowable input-output statements by file organization, access mode, and open mode.

Table 5-7: Allowable Input-Output Statements

File Organization	Access Mode	Statement	Open Mode			
			INPUT	OUTPUT	I-O	EXTEND
SEQUENTIAL	SEQUENTIAL	READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	No	Yes
RELATIVE	SEQUENTIAL	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	No	No
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	Yes	No
INDEXED	SEQUENTIAL	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	No	No
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	Yes	No

6. An executable image can open a file more than once with the INPUT, OUTPUT, I-O, and EXTEND phrases. After the first OPEN statement, each later OPEN for the same file must follow the execution of a CLOSE statement for the file. The CLOSE statement must not have a REEL, UNIT, or LOCK phrase.

7. The OPEN statement does not get or release the first data record.

8. For an OPEN statement with the INPUT, I-O, or EXTEND phrases, the file's File Description entry must be equivalent to that used when the file was created.

9. The NO REWIND phrase applies only to sequential single-reel/unit files. It has no effect if the concept of rewinding does not apply to the file's storage medium.

10. If the file's storage medium allows rewinding, and:

    a. There is neither an EXTEND nor NO REWIND phrase, then OPEN statement execution positions the file at its beginning.

    b. There is a NO REWIND phrase, then the OPEN statement does not reposition the file. The file must already be positioned at its beginning before the OPEN statement executes.

11. If the file opened with the INPUT phrase is an optional file that is not present, the OPEN statement sets the Next Record Pointer to indicate this condition.

12. For indexed files opened with the INPUT or I-O phrase, the OPEN statement sets the Next Record Pointer to the first record existing in the file when it is opened. The Prime Record Key is established as the Key of Reference. It determines the first record to be accessed. If the file has no records, the OPEN statement sets the Next Record Pointer to cause an at end condition on the next sequential READ statement for the file.

13. An OPEN statement with the EXTEND phrase positions the file immediately after its last logical record. The definition of last logical record differs by file organization:

    a. For sequential files, it is the last record written in the file.

    b. For relative files, it is the currently existing record with the highest relative record number.

    c. For indexed files, it is the currently existing record with the highest Prime Record Key value.

14. The I-O phrase opens a mass storage file for both input and output operations.

15. Successful execution of an OPEN statement with the EXTEND or I-O phrase creates the file if it is not available. Successful execution of an OPEN statement with the OUTPUT phrase creates the file. In each case, the created file contains no data records.

16. Successful execution of an OPEN statement sets the Current Volume Pointer to:
    a. The first or only reel/unit for an available input or input-output file
    b. The reel/unit containing the last logical record for an extend file
    c. The new reel/unit for an unavailable output, input-output, or extend file
17. If there is more than one file-name in the OPEN statement, execution is the same as if there were separate OPEN statements, one for each file-name.
18. The ALLOWING phrase specifies a file-sharing option for the file.
19. The NO OTHERS phrase specifies exclusive file access.
20. The ALL phrase specifies unlimited file sharing.
21. The READERS phrase allows access only by READ statement.
22. The WRITERS phrase allows access only by WRITE statement.
23. The UPDATERS phrase allows access only by READ, DELETE, and REWRITE statements.
24. If there is no ALLOWING phrase:
    a. The default for files in the input mode is ALLOWING READERS.
    b. The default for files in other than the input mode is ALLOWING NO OTHERS.

Technical Note

1. OPEN statement execution can result in these FILE STATUS data item values:

FILE STATUS	Meaning
00	Successful
05	Optional file not present
91	File is locked by another program
94	File is already open, or closed with lock
95	No file space on device
97	File not found
30	All other permanent errors

Additional References

Section 5.15 CLOSE Statement

5.31 READ Statement

Function

For sequential access files, the READ statement makes the next logical record available. For random access files, READ makes a specified record available.

General Format

Format 1

```
READ file-name [ NEXT ] RECORD [ INTO dest-item ]
    [ AT END stment [ END-READ ] ]
```

Format 2

```
READ file-name RECORD [ INTO dest-item ]
    [ KEY IS key-name ]
    [ INVALID KEY stment [ END-READ ] ]
```

file-name
is the name of a file described in the Data Division. It cannot be a sort or merge file.

dest-item
is the identifier of a data item that contains the record accessed by the READ statement.

stment
is an imperative statement executed for an at end or invalid key condition.

key-name
is the data-name of a data item specified as a Record Key for file-name. It can be qualified.

Syntax Rules

1. Format 1 must be used for a sequential access mode file.
2. There must be a NEXT phrase for dynamic access mode files to retrieve records sequentially.
3. Format 2 can be used for random or dynamic access mode files to retrieve records randomly.

4. The KEY phrase can be used only for indexed files.
5. There must be an INVALID KEY or AT END phrase when there is no applicable USE AFTER EXCEPTION procedure for the file.

General Rules

1. The file must be open in the INPUT or I-O mode when the READ statement executes.
2. For sequential access mode files, the NEXT phrase is optional. It has no effect on READ statement execution.
3. Executing a Format 1 READ statement can cause the following to occur:
   a. The record pointed to by the Next Record Pointer becomes available in the file's record area.
   b. For sequential and relative files, the Next Record Pointer points to the file's next existing record.
   c. For indexed files, the Next Record Pointer points to the next existing record established by the file's Key of Reference.
   d. If the file has no next record, the Next Record Pointer indicates that no next logical record exists.
4. The READ statement updates the value of the FILE STATUS data item for the file.
5. A record is available before any statement executes after the READ.
6. More than one record description can describe a file's logical records. The records then share the same record area in storage. Sharing a record area is equivalent to implicit redefinition.

   READ statement execution does not change the contents of data items in the record area beyond the range of the current data record. The contents are undefined.
7. A Format 1 READ statement can recognize the end of a reel/unit during its execution. If it has not reached the logical end of the file, the READ statement performs a reel/unit swap. The Current Volume Pointer points to the file's next reel/unit.
8. During execution of a Format 2 READ statement, the Next Record Pointer can indicate that an optional file is not present. The invalid key condition then exists, and READ statement execution is unsuccessful.
9. When a Format 1 READ statement executes, the Next Record Pointer can indicate that:
   - There is no next logical record

- No valid next record has been established
- An optional file is not present

When the READ statement detects one of these conditions:

a. It updates the FILE STATUS data item for the file to indicate the at end condition.

b. If the READ statement has an AT END phrase, control transfers to stment. No USE AFTER EXCEPTION procedure for the file executes.

c. If there is no AT END phrase, a USE AFTER EXCEPTION procedure must be associated with the file. Control transfers to that procedure. Control returns from the USE AFTER EXCEPTION procedure to the next executable statement after the end of the READ statement.

When the at end condition occurs, execution of the input-output statement that caused it is unsuccessful.

10. After the unsuccessful execution of a READ statement, the contents of the file's record area are undefined. If an optional file is not present, the Next Record Pointer is unchanged; otherwise, it indicates that no valid next record has been established. For indexed files, the Key of Reference is undefined.

11. For a relative or indexed file with dynamic access mode, a Format 1 READ statement with the NEXT phrase retrieves the file's next logical record.

12. For a relative file, a Format 1 READ statement updates the contents of the file's RELATIVE KEY data item. The data item contains the relative record number of the available record.

13. For a relative file, a Format 2 READ statement sets the Next Record Pointer to the record whose relative record number is in the file's RELATIVE KEY data item. Execution then continues as specified in General Rule 3.

    If the record is not in the file, the invalid key condition exists, and READ statement execution is unsuccessful.

14. For a sequentially accessed indexed file, records with an identical value in an Alternate Record Key that is the Key of Reference are made available in the same order the duplicate values were created. The duplicate values can be created by execution of WRITE or REWRITE statements.

15. For an indexed file, a Format 2 READ statement with the KEY phrase establishes key-name as the Key of Reference for the retrieval. For a dynamic access mode file, the same Key of Reference applies to later retrievals by Format 1 READ statement executions for the file. The Key of Reference continues in effect until a new Key of Reference is established.

16. For an indexed file, a Format 2 READ statement without the KEY phrase establishes the Prime Record Key as the Key of Reference for the retrieval. For a dynamic access mode file, the same Key of Reference applies to later retrievals by Format 1 READ statement executions for the file. The Key of Reference continues in effect until a new Key of Reference is established.

17. For an indexed file, a Format 2 READ statement compares the value in the Key of Reference with the value in the corresponding data item in the file's records. The comparison continues until the READ statement finds the first record with an equal value. For an alternate key with duplicate values, the first record found is the first of a sequence of duplicates released to RMS. The READ statement sets Next Record Pointer to the record. Execution then continues as specified in General Rule 3.

    If the READ statement cannot identify a record with an equal value, the invalid key condition exists. READ statement execution is then unsuccessful.

    If the size of the retrieved record exceeds the maximum size specified for the file, READ statement execution is unsuccessful.

If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input condition occurs that would result in a nonzero value in a FILE STATUS data item. However, it does not execute if: (1) the condition is *invalid key* and there is an INVALID KEY phrase or (2) the condition is *at end* and there is an AT END phrase.

Technical Note

READ statement execution can result in these FILE STATUS data item values:

FILE STATUS	File Organization	Access Method	Meaning
00	All	All	Successful.
13	All	Seq	No next logical record (at end).
15	All	Seq	Optional file not present (at end).
16	All	Seq	No valid next record (at end).
23	Ind, Rel	Rand	Record not in file (invalid key).
25	Ind, Rel	Rand	Optional file not present (invalid key).
90	All	All	Record locked by another program; record available in record area.
92	All	All	Record locked by another program.
94	All	All	File not open or incompatible open mode.
30	All	All	All other permanent errors.

READ
(Continued)

Additional References

Section 5.3	Scope of Statements
Section 5.8.9	I-O Status
Section 5.8.10	Invalid Key Condition
Section 5.8.13	INTO Option
Section 5.29	OPEN Statement

5.32 RELEASE Statement

Function

The RELEASE statement transfers records to a SORT operation.

General Format

```
RELEASE rec [ FROM src-area ]
```

rec
is the name of a logical record in a Sort-Merge File Description (SD) entry. It can be qualified.

src-area
is the identifier of the data item that contains the data.

Syntax Rule

A RELEASE statement can be used only in an input procedure. The input procedure must be associated with a SORT statement for the Sort-Merge file that contains rec.

General Rules

1. The description of the FROM phrase appears in the FROM Option entry.
2. The RELEASE statement transfers the contents of **rec** to the first phase of the sort.
3. After the RELEASE statement executes, the record is no longer available in **rec** unless the associated sort-merge file-name is in a SAME RECORD AREA clause. In that case, the record is available to the program as a record of the sort-merge file-name. It is also available as a record of all other file-names in the same SAME RECORD AREA clause.

Additional References

Section 5.8.12	FROM Option
VAX-11 COBOL User's Guide	Sorting and Merging

5.33 RETURN Statement

Function

The RETURN statement gets sorted records from a SORT operation. It also returns merged records in a MERGE operation.

General Format

```
RETURN smrg-file RECORD [ INTO dest-area ]
  AT END stment
  [ END-RETURN ]
```

smrg-file
is the name of a file described in a Sort-Merge File Description (SD) entry.

dest-area
is the identifier of the data item to which the returned smrg-file record is moved.

stment
is an imperative statement.

Syntax Rule

A RETURN statement can be used only in an output procedure. The output procedure must be associated with a SORT or MERGE statement for smrg-file.

General Rules

1. The description of the INTO phrase appears in the INTO Option entry.
2. When more than one record description describes the logical records for smrg-file, the records share the same storage area. The contents of storage positions beyond the range of the returned record are undefined when the RETURN statement ends.
3. Before the output procedure executes, the Next Record Pointer is updated. It points to the record whose key values make it first in the file. If there are no records, the Next Record Pointer indicates the at end condition.
4. The RETURN statement makes the next record (pointed to by the Next Record Pointer) available in the record area for smrg-file.
5. The Next Record Pointer is updated to point to the next record in smrg-file. The key values in the SORT or MERGE statement determine which is the next record.

6. If smrg-file has no next record, the Next Record Pointer is updated to indicate the at end condition.

7. If the Next Record Pointer indicates the at end condition when the RETURN statement executes, control transfers to stment. The contents of the smrg-file record areas are then undefined.

8. When the at end condition occurs:

   - RETURN statement execution is unsuccessful.
   - The Next Record Pointer is not changed.

9. The description of the END-RETURN phrase appears in the entry for Scope of Statements.

Additional References

Section 5.3	Scope of Statements
Section 5.8.13	INTO Option
VAX-11 COBOL User's Guide	Sorting and Merging

5.34 REWRITE Statement

Function

The REWRITE statement logically replaces a mass storage file record.

General Format

```
REWRITE rec-name [ FROM src-item ]

  [ INVALID KEY stment [ END-REWRITE ] ]
```

rec-name
is the name of a logical record in the Data Division File Section. It can be qualified. The logical record cannot be in a Sort-Merge File Description Entry.

src-item
is the identifier of the data item that contains the data.

stment
is an imperative statement.

Syntax Rules

1. The INVALID KEY phrase cannot be used in a REWRITE statement that refers to a sequential or relative file with sequential access mode.
2. For a relative file with random or dynamic access mode or for an indexed file, the REWRITE statement must have an INVALID KEY phrase when there is no applicable USE AFTER EXCEPTION procedure for the file.

General Rules

All Files

1. The file associated with rec-name must be a mass storage file. It must be open in the I-O mode when the REWRITE statement executes.
2. For sequential access mode files, the last input-output statement executed for the file before the REWRITE statement must be a successfully executed READ or START. The REWRITE statement logically replaces the record accessed by the READ or positioned by the START.
3. The record is no longer available in rec-name after a REWRITE statement successfully executes. However, if the associated file-name is in a SAME RECORD AREA clause, the record is available in rec-name. It is also available in the record areas of other file-names in the same SAME RECORD AREA clause.

4. The REWRITE statement does not affect the Next Record Pointer.

5. The REWRITE statement updates the value of the FILE STATUS data item for the file.

Sequential Files

6. The record named by rec-name must be the same size as the record being replaced.

Relative Files

7. For a random or dynamic access mode file, the REWRITE statement logically replaces the record specified in the RELATIVE KEY data item for rec-name's file. If the record is not in the file, the invalid key condition exists. The update does not occur, and the data in the record area is not affected.

Indexed Files

8. For a sequential access mode file, the Prime Record Key specifies the record to be replaced. The values of the Prime Record Keys in the record to be replaced and the last record read from (or positioned in) the file must be equal.

9. For a random or dynamic access mode file, the Prime Record Key specifies the record to replace.

10. For a record with an Alternate Record Key:

    a. When the REWRITE does not change the value of an Alternate Record Key, the order of retrieval is unchanged when the key is the Key of Reference.

    b. When duplicate key values are allowed and the value of an Alternate Record Key changes, the later retrieval order of the record changes when the key is the Key of Reference. The record's logical position is last in the group of records with the same value in the Alternate Record Key that changed.

11. Any of the following cause the invalid key condition:

    a. The access mode is sequential, and the values in the Prime Record Keys of the record to replace and the last record read from (or positioned in) the file are not equal.

    b. The value in the Prime Record Key does not equal that of any record in the file.

    c. The value in an Alternate Record Key whose definition does not have a DUPLICATES clause equals that of a record already in the file.

    The update does not occur, and the data in the record area is not affected.

If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a non-zero value in a FILE STATUS data item. However, it does not execute if the condition is invalid key, and there is an INVALID KEY phrase.

Technical Note

REWRITE statement execution can result in these FILE STATUS data item values:

FILE STATUS	File Organization	Access Method	Meaning
00	All	All	Successful
02	Ind	All	Created duplicate Alternate Key
21	Ind	Seq	Primary key changed after READ or START (invalid key)
22	Ind	All	Duplicate Alternate Key (invalid key)
23	Ind, Rel	Rand	Record not in file (invalid key)
92	Ind, Rel	All	Record locked by another program
93	All	Seq	No previous READ or START
94	All	All	File not open, or incompatible open mode
30	All	All	All other permanent errors

Additional References

Section 5.3 Scope of Statements
Section 5.8.9 I-O Status
Section 5.8.10 Invalid Key Condition
Section 5.8.12 FROM Option
Section 5.29 OPEN Statement

5.35 SEARCH Statement

Function

The SEARCH statement searches for a table element that satisfies a condition. It sets the value of the associated index to point to the table element.

General Format

Format 1

```
SEARCH src-table [ VARYING pointr ]
    [ AT END stment ]
  { WHEN cond { stment          } } ... [ END-SEARCH ]
  {           { NEXT SENTENCE  } }
```

Format 2

```
SEARCH ALL src-table [ AT END stment ]
    WHEN { elemnt    { IS EQUAL TO } }  arg }
         {           { IS =        } }     }
         { cond-name                       }
  [ AND { elemnt    { IS EQUAL TO } }  arg } ] ...
        {           { IS =        } }     }
        { cond-name                       }
      { stment         }
      { NEXT SENTENCE  }  [ END-SEARCH ]
```

src-table
is an identifier that identifies the table.

pointr
is the identifier of a data item described as USAGE INDEX or an elementary numeric data item with no positions to the right of the assumed decimal point.

(continued on next page)

cond
is any conditional expression.

stment
is an imperative statement.

elemnt
is an indexed data-name. It refers to the table element against which the argument is compared.

arg
is the argument tested against each elemnt in the search. It is an identifier, literal, or arithmetic expression.

cond-name
is a condition-name.

Syntax Rules

Both Formats

1. Src-table must not be subscripted, indexed, or reference-modified. However, its description must contain an OCCURS clause with the INDEXED BY phrase.
2. The END-SEARCH and NEXT SENTENCE phrases cannot be used in the same SEARCH statement.

Format 2

3. Src-table must contain the KEY IS phrase in its OCCURS clause.
4. Each cond-name must be defined as having only one value. The data-name associated with cond-name must be in the KEY IS phrase of the OCCURS clause for src-table.
5. Each elemnt:
   - Can be qualified
   - Must be indexed by the first index-name associated with src-table, in addition to other indexes or literals required for uniqueness
   - Must be in the KEY IS phrase of the OCCURS clause for src-table
6. Neither arg nor any identifier in its arithmetic expression can:
   - Be used in the KEY IS phrase of the OCCURS clause for src-table
   - Be indexed by the first index-name associated with src-table

7. When the data-name associated with cond-name or elemnt is in the KEY phrase of the OCCURS clause for src-table, each preceding data-name (or associated cond-name) in that phrase must also be referenced.

General Rules

Both Formats

1. After the execution of a stment that does not end with a GO TO statement, control passes to the end of the SEARCH statement.

2. Src-table can be subordinate to a data item that contains an OCCURS clause. In that case, an index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. The SEARCH statement modifies the setting of only the index-name for src-table (and pointr, if there is one).

   A SEARCH statement must execute several times to search a multi-dimensional table. Before each execution, SET statements must execute to change the values of index-names that need adjustment.

Format 1

3. The Format 1 SEARCH statement searches a table serially, starting with the current index setting.

   a. The index-name associated with src-table can contain a value that indicates a higher occurrence number than is allowed for src-table. If the SEARCH statement execution starts when this condition exists, the search terminates immediately. If there is an AT END phrase, stment then executes. Otherwise, control passes to the end of the SEARCH statement.

   b. If the index-name associated with src-table indicates a valid src-table occurrence number, the SEARCH statement evaluates the conditions in the order they appear. It uses the index settings to determine the occurrence numbers of items to test.

      If no condition is satisfied, the index-name for src-table is incremented to refer to the next occurrence. The condition evaluation process repeats using the new index-name settings. However, if the new value of the index-name for src-table indicates a table element outside its range, the search terminates as in General Rule 3a.

      When a condition is satisfied:

      - The search terminates immediately.
      - The stment associated with the condition executes.
      - The index-name remains set at the occurrence that satisfied the condition.

4. If there is no VARYING phrase, the index-name used for the search is the first index-name in the OCCURS clause for src-table. Other src-table index-names are unchanged.

5. The VARYING phrase pointr can be used in the INDEXED BY phrase of the OCCURS clause for src-table. The search then uses that index-name. Otherwise, it uses the first index-name in the INDEXED BY phrase.

6. The VARYING phrase pointr can be used in the INDEXED BY phrase in the OCCURS clause for *another* table entry. In that case, the search increments the occurrence number represented by pointr by the same amount, and at the same time, as it increments the occurrence number represented by the src-table index-name.

7. If the VARYING phrase pointr is an index data item rather than an index-name, the search increments it by the same amount, and at the same time, as it increments the src-table index-name. If the VARYING phrase pointr is not an index data item or an index-name, the search increments it by one when it increments the src-table index-name.

8. Figure 5–6 describes the operation of a Format 1 SEARCH statement with two WHEN phrases.

Format 2

9. A SEARCH ALL operation yields predictable results only when both:

   - The data in the table has the same order as described in the KEY IS phrase of the OCCURS clause for src-table
   - The contents of the keys in the WHEN phrase identify a unique table element.

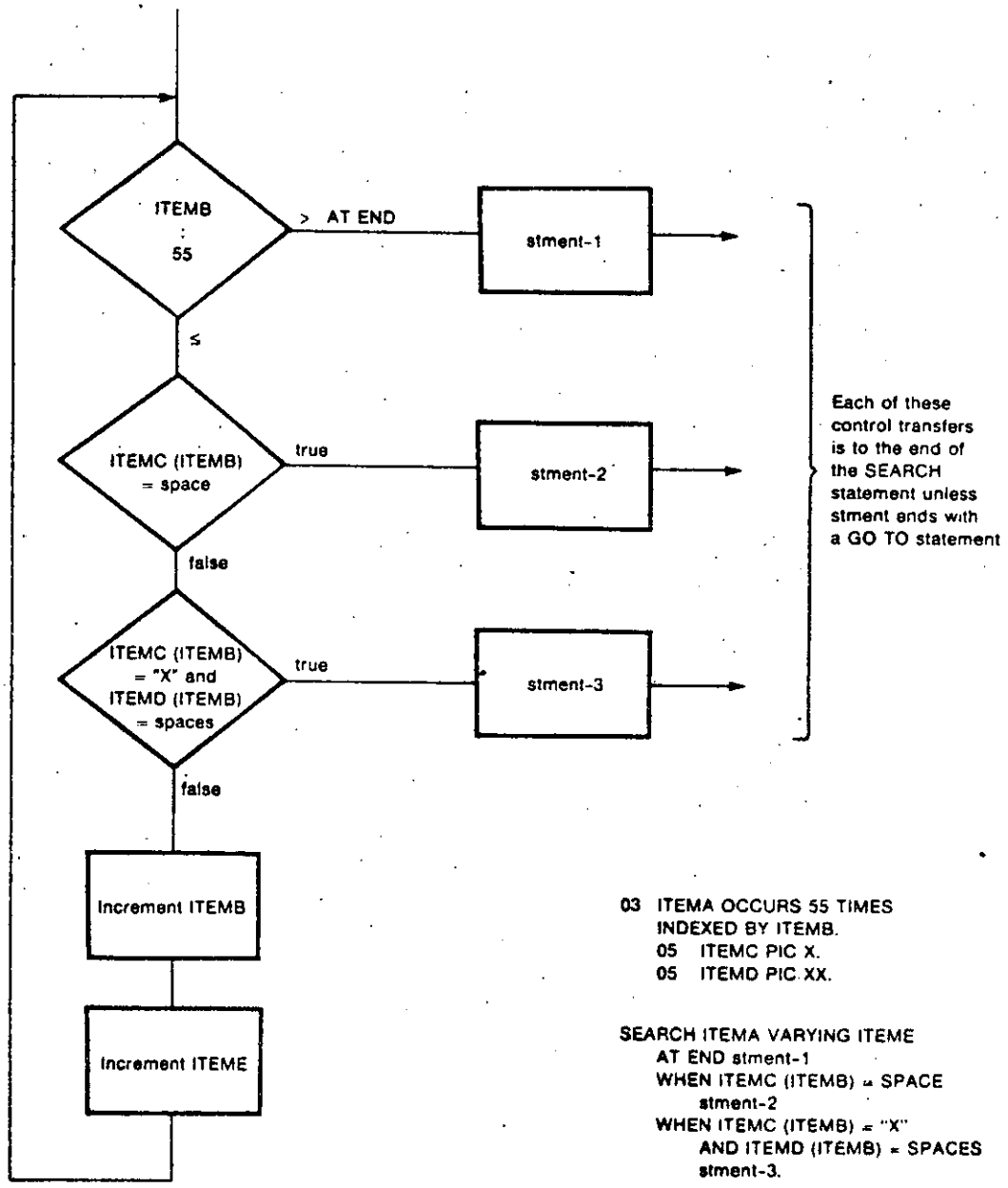
10. SEARCH ALL causes a nonserial, or binary, search. It ignores the initial setting of the src-table index-name and varies its setting during execution.

11. If the WHEN phrase conditions are not satisfied for any index setting in the allowed range, control passes to the AT END phrase stment, if there is one, or to the end of the SEARCH statement. In either case, the setting of the src-table index-name is not predictable.

12. If all the WHEN phrase conditions are satisfied for an index setting in the allowed range, control passes to either stment or the next sentence, whichever is in the statement. The src-table index-name then indicates the occurrence number that satisfied the conditions.

13. The index-name used for the search is the first index-name in the OCCURS clause for src-table. Other src-table index-names are unchanged.

Figure 5–6: Format 1 SEARCH Statement with Two WHEN Phrases



F-MK-00192-00

Additional References

Section 4.4.6	OCCURS Clause
Section 5.3	Scope of Statements
Section 5.7	Conditional Expressions

Examples

The examples assume these Data Division entries:

```
01  CUSTOMER-REC.
    03  CUSTOMER-USPS-STATE  PIC XX.
    03  CUSTOMER-REGION      PIC X.
    03  CUSTOMER-NAME        PIC X(15).

01  STATE-TAB.
    03  FILLER  PIC X(153) VALUE
          "AK3AL5AR5AZ4CA4CO4CT1DC1DE1FL5GA5HI3
-         "IA2ID3IL2IN2KS2KY5LA5MA1MD1ME1MI2MN2
-         "MO5MS5MT3NC5ND3NE2NH1NJ1NM4NV4NY1OH2
-         "OK4OR3PA1RI1SC5SD3TN5TX4UT4VA5VT1WA3
-         "WI2WV5WY4".
01  STATE-TABLE REDEFINES STATE-TAB.
    03  STATES OCCURS 51 TIMES
        ASCENDING KEY IS STATE-USPS-CODE
        INDEXED BY STATE-INDEX.
        05  STATE-USPS-CODE  PIC XX.
        05  STATE-REGION     PIC X.

01  STATE-NUM   PIC 99.
01  STATE-ERROR PIC 9.

01  NAME-TABLE VALUE SPACES.
    03  NAME-ENTRY OCCURS 8 TIMES
        INDEXED BY NAME-INDEX.
        05        LAST-NAME    PIC X(15).
        05        NAME-COUNT   PIC 999.
```

1. Binary search. The correctness of this statement's operation depends on the ascending order of key values.

```
SEARCH ALL STATES
  AT END
    MOVE 1 TO STATE-ERROR
  WHEN STATE-USPS-CODE (STATE-INDEX) = CUSTOMER-USPS-STATE
    MOVE 0 TO STATE-ERROR
    MOVE STATE-REGION (STATE-INDEX) TO CUSTOMER-REGION.
```

Results

CUSTOMER-STATE	CUSTOMER-REGION	STATE-INDEX	STATE-ERROR
NH	1	31	0
CA	4	5	0
DM		10	1
WY	4	51	0

2. Serial search with two WHEN phrases. The initial value of CUSTOMER-REGION is 2.

```
SEARCH-LOOP.
    SEARCH STATES
      AT END MOVE 1 TO STATE-ERROR
      WHEN STATE-REGION (STATE-INDEX) = CUSTOMER-REGION
        NEXT SENTENCE
      WHEN STATE-USPS-CODE (STATE-INDEX) = "NH"
        MOVE 3 TO STATE-ERROR.
    SET STATE-NUM TO STATE-INDEX.
    DISPLAY STATE-USPS-CODE (STATE-INDEX) " "
      STATE-NUM " " STATE-ERROR.
    IF STATE-ERROR NOT = 1
      SET STATE-INDEX UP BY 1
      GO TO SEARCH-LOOP.
```

Results

```
IA    13    0
IL    15    0
IN    16    0
KS    17    0
MI    23    0
MN    24    0
NE    30    0
NH    31    3
OH    36    3
WI    49    3
      52    1
```

3. Updating a table in a SEARCH statement.

```
GET-NAME.
    DISPLAY "Enter name: " NO ADVANCING.
    ACCEPT CUSTOMER-NAME.
    SET NAME-INDEX TO 1.
    SEARCH NAME-ENTRY
      AT END
        DISPLAY "   Table full".
        SET NAME-INDEX TO 1
        PERFORM SHOW-TABLE 8 TIMES
        STOP RUN
      WHEN LAST-NAME (NAME-INDEX) = CUSTOMER-NAME
        ADD 1 TO NAME-COUNT (NAME-INDEX)
      WHEN LAST-NAME (NAME-INDEX) = SPACES
        MOVE CUSTOMER-NAME TO LAST-NAME (NAME-INDEX)
        MOVE 1 TO NAME-COUNT (NAME-INDEX).
    GO TO GET-NAME.
SHOW-TABLE.
    DISPLAY LAST-NAME (NAME-INDEX) "
        " NAME-COUNT (NAME-INDEX).
    SET NAME-INDEX UP BY 1.
```

Results

```
Enter name: SMITH
Enter name: JONES
Enter name: SMITH
Enter name: THOMPSON
Enter name: MACINTOSH
Enter name: SMITH
Enter name: JAMES
Enter name: FRIED
Enter name: ADAMS
Enter name: MACINTOSH
Enter name: SMITH
Enter name: JAMES
Enter name: MACINTOSH
Enter name: KAPLAN
Enter name: WILLIAMS
     Table full
SMITH          004
JONES          001
THOMPSON       001
MACINTOSH      003
JAMES          002
FRIED          001
ADAMS          001
KAPLAN         001
```

82

5.36 SET Statement

Function

The SET statement sets values of indexes associated with table elements. It can also change the value of a conditional variable and the status of an external switch.

General Format

Format 1

```
SET  | rsult | ...  TO  val
```

Format 2

```
SET  | indx | ...  { UP BY   }  increm
                  { DOWN BY }
```

Format 3

```
SET  | cond-name | ...  TO TRUE
```

Format 4

```
SET  { | switch-name | ...  TO  { ON  } } ...
                               { OFF }
```

rsult
is an index-name or the identifier of an index data item or an elementary numeric integer data item.

val
is a positive integer, which may be signed. It can also be an index-name or the identifier of an index data item or an elementary numeric integer data item.

indx
is an index-name.

increm
is an integer, which may be signed. It can also be the identifier of an elementary numeric integer data item.

(continued on next page)

cond-name
is a condition-name that must be associated with a conditional variable.

switch-name
is the name of an external switch defined in the SPECIAL-NAMES paragraph.

Syntax Rule

No two occurrences of cond-name can refer to the same conditional variable.

General Rules

Formats 1 and 2

1. Index-names are associated with a table in the table's OCCURS clause INDEXED BY phrase.
2. If rsult is an index-name, its value after SET statement execution must correspond to an occurrence number of an element in the associated table.
3. If val is an index-name, its value before SET statement execution must correspond to an occurrence number of an element in the table associated with rsult.
4. The value of indx, both before and after SET statement execution, must correspond to an occurrence number of an element in the table associated with indx.

Format 1

5. The SET statement sets the value of rsult to refer to the table element whose occurrence number corresponds to the table element referred to by val. If val is an index data item, no conversion occurs.
6. If rsult is an index data item, val cannot be an integer. No conversion occurs when rsult is set to the value of val.
7. If rsult is not an index data item or an index-name, val can only be an index-name.
8. When there is more than one rsult, SET uses the original value of val in each operation. Subscript or index evaluation for rsult occurs immediately before its value changes.
9. Table 5-8 shows the validity of operand combinations. An asterisk (*) means that no conversion occurs during the SET operation.

Table 5-8: Validity of Operand Combinations in Format 1 SET Statements

Sending Item	Receiving Item		
	Integer Data Item	Index	Index Data Item
Integer Literal	Invalid/Rule 7	Valid/Rule 5	Invalid/Rule 6
Integer Data Item	Invalid/Rule 7	Valid/Rule 5	Invalid/Rule 6
Index	Valid/Rule 7	Valid/Rule 5	Valid/Rule 6*
Index Data Item	Invalid/Rule 7	Valid/Rule 5*	Valid/Rule 6*

Format 2

10. The SET statement increments (UP) or decrements (DOWN) indx by a value that corresponds to the number of occurrences increm represents.

11. When there is more than one indx, SET uses the original value of increm in each operation.

Format 3

12. SET moves the literal in the VALUE clause for cond-name to its associated conditional variable. The transfer occurs according to the rules for elementary moves. If the VALUE clause contains more than one literal, the first is moved.

Format 4

13. SET changes the status of each switch-name in the statement.

14. The ON phrase changes the status of switch-name to "on."

15. The OFF phrase changes the status of switch-name to "off."

16. The SET statement changes the switch status only for the image in which it executes. When the image terminates, the status of each external switch is the same as when the image began.

Additional References

Section 3.1.3	SPECIAL-NAMES Paragraph
Section 5.7.8	Switch-Status Condition
Section 5.27	MOVE Statement
Section 5.30	PERFORM Statement
Section 5.35	SEARCH Statement

Examples

The SEARCH statement examples show the use of Format 1 and Format 2 SET statements.

The examples assume these Environment and Data Division entries:

```
SPECIAL-NAMES.
    SWITCH 1 UPDATE-RUN ON STATUS IS DO-UPDATE
    SWITCH 3 REPORT-RUN ON STATUS IS DO-REPORT
        OFF STATUS IS SKIP-REPORT
    SWITCH 4 IS NEW-YEAR ON STATUS IS BEGIN-YEAR
        OFF IS CONTINUE-YEAR.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  YEAR-LEVEL  PIC 99.
    88  FRESHMAN VALUE 1.
    88  SOPHOMORE VALUE 2.
    88  JUNIOR VALUE 3.
    88  SENIOR VALUE 4.
    88  FIRST-MASTERS VALUE 5.
    88  MASTERS VALUE 5,6.
    88  FIRST-DOCTORAL VALUE 7.
    88  DOCTORAL VALUE 7,8.
    88  NON-DEGREE-UNDERGRAD VALUE 9.
    88  NON-DEGREE-GRAD VALUE 10.
    88  UNDERGRAD VALUE 9, 1 THROUGH 4.
    88  GRAD VALUE 10, 5 THROUGH 8.
```

		YEAR-LEVEL
1.	SET SOPHOMORE TO TRUE	02
2.	SET MASTERS TO TRUE	05
3.	SET GRAD TO TRUE	10
4.	SET NON-DEGREE-GRAD TO TRUE	10

5. Setting external switches. The truth value shows the result of the IF statements:

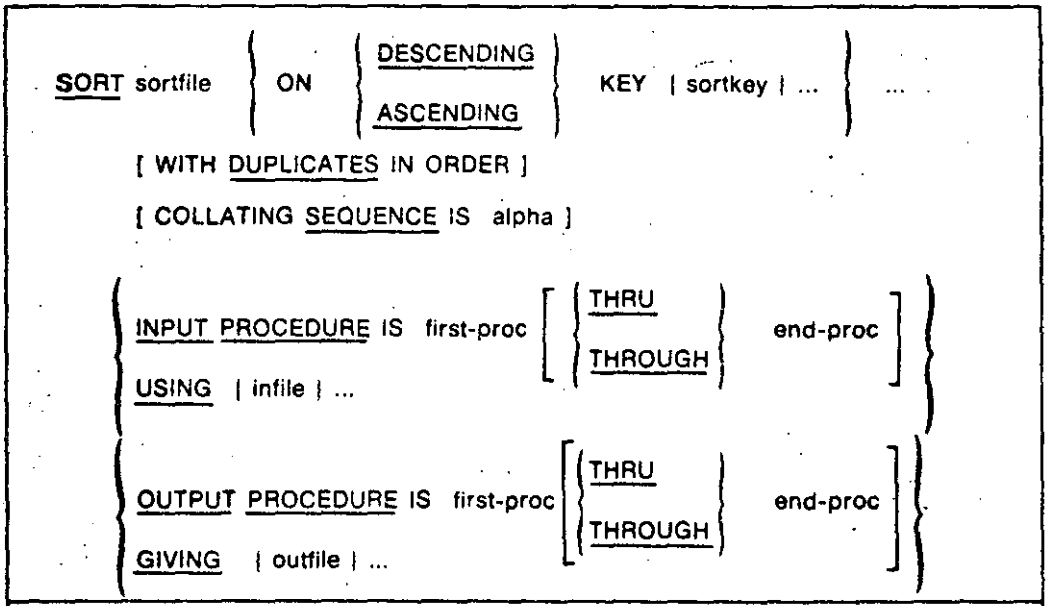
```
SET UPDATE-RUN TO ON.
SET REPORT-RUN TO OFF.
SET NEW-YEAR TO ON.
IF DO-UPDATE ...                    Truth Value
                                    true
IF DO-REPORT ...                    false
IF CONTINUE-YEAR ...                false
SET REPORT-RUN TO ON.
IF DO-REPORT ...                    true
IF SKIP-REPORT ...                  false
```

5.37 SORT Statement

Function

The SORT statement creates a sort file by executing input procedures or transferring records from an input file. It sorts the records in the sort file on a set of keys. Finally, it returns each record from the sort file, in sorted order, to output procedures or an output file.

General Format



sortfile
is a file-name described in a Sort-Merge File Description (SD) entry in the Data Division.

sortkey
is the data-name of a data item in a record associated with sortfile.

alpha
is an alphabet-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

first-proc
is the section-name of the first section of the input or output procedure.

infile
is the file-name of the input file. It must be described in a File Description (FD) entry in the Data Division.

(continued on next page)

end-proc
is the section-name of the last section of the input or output procedure.

outfile
is the file-name of the output file. It must be described in a File Description (FD) entry in the Data Division.

Syntax Rules

1. SORT statements can be used anywhere in the Procedure Division except in:
   a. Declaratives
   b. A SORT or MERGE statement input or output procedure
2. If sortfile contains variable-length records, infile records must not be smaller than the smallest in sortfile nor larger than the largest.
3. If sortfile contains fixed-length records, infile records must not be larger than the largest record described for sortfile.
4. If outfile contains variable-length records, sortfile records must not be smaller than the smallest in outfile nor larger than the largest.
5. If outfile contains fixed-length records, sortfile records must not be larger than the largest record described for outfile.
6. Sortkey can be qualified.
7. Sortkey cannot be a group that contains variable-occurrence data items.
8. The sortkey description cannot contain an OCCURS clause or be subordinate to a Data Description entry that does.
9. Sortfile can have more than one record description. However, sortkey need be described in only one of the record descriptions. The character positions referenced by sortkey are used as the key for all the file's records.
10. The words THRU and THROUGH are equivalent.
11. If outfile is an indexed file, the first sortkey must be in the ASCENDING phrase. It must specify the same character positions in its record as the prime record key for outfile.

General Rules

1. If sortfile contains fixed-length records, any shorter infile records are space-filled on the right after the last character. Space-filling occurs before the infile record is released to sortfile.

2. The leftmost sortkey is the major key, and the next sortkey is the next most significant key. The significance of sortkey data items is not affected by how they are divided into KEY phrases. Only left-to-right order determines significance.

3. The ASCENDING phrase causes the sorted sequence to be from the lowest sortkey value to the highest.

4. The DESCENDING phrase causes the sorted sequence to be from the highest sortkey value to the lowest.

5. Sort sequence follows the rules for relation condition comparisons.

6. The DUPLICATES phrase affects the return order of records whose corresponding sortkey values are equal.

   a. When there is a USING phrase, return order is the same as the order of appearance of infile names in the SORT statement.

   b. When there is an input procedure, return order is the same as the order in which the records were released.

7. If there is no DUPLICATES phrase, the return order is undefined for records with equal corresponding sortkey values.

8. The SORT statement determines the comparison collating sequence for nonnumeric sortkey items when it begins execution. If there is a COLLATING SEQUENCE phrase in the SORT statement, SORT uses that sequence. Otherwise, it uses the program collating sequence.

9. The input procedure consists of one or more sections that:

   - Appear contiguously in the source program
   - Do not form a part of any output procedure

10. The input procedure must execute at least one RELEASE statement to transfer records to sortfile.

11. The program must not pass control to the input procedure except during execution of a related SORT statement.

12. The input procedure cannot contain SORT or MERGE statements. It must not explicitly transfer control outside the input procedure. However, statements can cause implied control transfers to Declaratives.

13. The remainder of the Procedure Division must not transfer control to points in the input procedure.

14. If there is an input procedure, control passes to it before the SORT statement sequences the sortfile records. When control passes the last statement in the input procedure's last section, the records released to sortfile are sorted.

15. During execution of the input or output procedures, or any USE AFTER EXCEPTION procedure implicitly invoked during the SORT statement, no statement can manipulate the files or record areas associated with infile or outfile.

16. If there is a USING phrase, the SORT statement transfers all records in infile to sortfile. This transfer is an implied SORT statement input procedure. When the SORT statement executes, infile must not be open.

17. For each infile, the SORT statement:

    a. Initiates file processing as if the program had executed an OPEN statement with the INPUT phrase.

    b. Gets the logical records and releases them to the sort operation. SORT obtains each record as if the program had executed a READ statement with the NEXT and AT END phrases.

    c. Terminates file processing as if the program had executed a CLOSE statement with no optional phrases. The SORT statement ends file processing before it executes any output procedure.

    These implicit OPEN, READ, and CLOSE operations can cause associated USE procedures to execute.

18. The output procedure consists of one or more sections that:

    - Appear contiguously in the source program
    - Do not form a part of any input procedure

19. When the SORT statement enters the output procedure, it is ready to select the next record in sorted order. The output procedure must execute at least one RETURN statement to make records available for processing.

20. The program must not pass control to the output procedure except during execution of a related SORT statement..

21. The output procedure cannot contain SORT or MERGE statements. It must not explicitly transfer control outside the output procedure. However, statements can cause implied control transfers to Declaratives.

22. The remainder of the Procedure Division must not transfer control to points in the output procedure.

23. If there is an output procedure, control passes to it after the SORT statement sequences the records in sortfile. When control passes the last statement in the output procedure's last section, the SORT statement ends. Control then transfers to the next executable statement after the SORT statement.

24. If there is a GIVING phrase, the SORT statement writes all sorted records to each outfile. This transfer is an implied SORT statement output procedure. When the SORT statement executes, outfile must not be open.

25. The SORT statement initiates outfile processing as if the program had executed an OPEN statement with the OUTPUT phrase. The SORT statement does not initiate outfile processing until after input procedure execution.

26. The SORT statement gets the sorted logical records and writes them to each outfile. SORT writes each record as if the program had executed a WRITE statement with no optional phrases.

    For relative files, the value of the relative key data item is 1 for the first returned record, 2 for the second, and so on. When the SORT statement ends, the value of the relative key data item indicates the number of outfile records.

27. The SORT statement terminates outfile processing as if the program had executed a CLOSE statement with no optional phrases.

28. These implicit OPEN, WRITE and CLOSE operations can cause associated USE procedures to execute. If the SORT statement tries to write beyond the boundaries of outfile, the applicable USE AFTER EXCEPTION procedure executes. If control returns from the USE procedure, or if there is none, outfile processing terminates as if the program had executed a CLOSE statement with no optional phrases.

29. If outfile contains fixed-length records, any shorter sortfile records are space-filled on the right after the last character. Space-filling occurs before the sortfile record is released to outfile.

30. If the SORT statement is in a *fixed* segment, its input and output procedures must be completely in either:

    a. Fixed segments

    b. One independent segment

31. If the SORT statement is in an *independent* segment, its input and output procedures must be completely in either:

    a. Fixed segments

    b. The same independent segment as the SORT statement itself

Additional References

Section 3.1.2	OBJECT-COMPUTER Paragraph
Section 3.1.3	SPECIAL-NAMES Paragraph
Section 3.4	I-O-CONTROL Paragraph
Section 5.9	Segmentation
VAX-11 COBOL User's Guide	Sorting and Merging

5.38 START Statement

Function

The START statement establishes the logical position in an indexed or relative file. The logical position affects subsequent sequential record retrieval.

General Format

```
START file-name [ KEY { IS EQUAL TO
                      IS =
                      IS GREATER THAN
                      IS >
                      IS NOT LESS THAN
                      IS NOT < } key-data ]

  [ INVALID KEY stment    [ END-START ] ]
```

file-name
is the name of an indexed or relative file with sequential or dynamic access. It cannot be the name of a sort or merge file.

key-data
is the data-name of a record key, or the leftmost part of a record key, for file-name. It can be qualified.

stment
is an imperative statement.

Syntax Rules

1. There must be an INVALID KEY phrase if file-name does not have an applicable USE FOR EXCEPTION procedure.
2. For a relative file, key-data must be the file's RELATIVE KEY data item.
3. For an indexed file, key-data can be either:
   - A record key for the file.
   - An alphanumeric data item subordinate to the description of the file's record key. The leftmost character position of key-data must correspond to that of the record key data item.

General Rules

All Files

1. The file must be open in the INPUT or I-O mode when the START statement executes.
2. If there is no KEY phrase, the implied relational operator is "EQUAL."
3. START statement execution does not change: (1) the contents of the record area or (2) the contents of the data item referred to in the DEPENDING ON phrase of the file's RECORD clause.
4. The comparison specified by the KEY phrase relational operator occurs between a key for a record in the file and a data item. (See General Rules 7, 8 , and 9.) If the file is indexed and the operand sizes are unequal, the comparison operates as if the longer one was truncated on the right to the size of the shorter. All other numeric or nonnumeric comparison rules apply.

   The Next Record Pointer is set to the first logical record in the file whose key satisfies the comparison.

   If no record in the file satisfies the comparison:

   - The invalid key condition exists.
   - START statement execution is unsuccessful.
   - The Next Record Pointer indicates that no valid next record is established.

5. The START statement updates the FILE STATUS data item for the file.
6. If the Next Record Pointer indicates that an optional file is not present when the START statement executes, the invalid key condition exists. START statement execution is then unsuccessful.

Relative Files

7. The comparison described in General Rule 4 uses the data item referred to by the RELATIVE KEY phrase in the file's ACCESS MODE clause.

Indexed Files

8. The START statement establishes a Key of Reference as follows:

   a. If there is no KEY phrase, the file's Prime Record Key becomes the Key of Reference.

   b. If there is a KEY phrase and key-data is a Record Key for the file, that Record Key becomes the Key of Reference.

c. If there is a KEY phrase and key-data is not a Record Key for the file, the Record Key whose leftmost character corresponds to the leftmost character of key-data becomes the Key of Reference.

The Key of Reference establishes the record ordering for the START statement. (See General Rule 4.) If the execution of the START statement is successful, later sequential READ statements use the same Key of Reference.

9. If there is a KEY phrase, the comparison described in General Rule 4 uses the contents of key-data.

10. If there is no KEY phrase, the comparison described in General Rule 4 uses the data item referred to in the file's RECORD KEY clause.

11. If START statement execution is not successful, the Key of Reference is undefined.

If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a non-zero value in a FILE STATUS data item. However, it does not execute if the condition is *invalid key* and there is an INVALID KEY phrase.

Technical Note

START execution can result in these FILE STATUS data item values:

FILE STATUS	Meaning
00	Successful
23	Record not in file (invalid key)
25	Optional file not present (invalid key)
92	Record locked by another program
94	File not open or incompatible open mode
30	All other permanent errors

Additional References

Section 5.3	Scope of Statements
Section 5.7.3	Comparison of Numeric Operands
Section 5.7.4	Comparison of Nonnumeric Operands
Section 5.8.9	I-O Status
Section 5.8.10	Invalid Key Condition
Section 5.29	OPEN Statement
Section 5.31	READ Statement

5.39 STOP Statement

Function

The STOP statement permanently or temporarily suspends image execution.

General Format

```
STOP { RUN  }
     { disp }
```

disp
is any literal except a figurative constant of the "ALL literal" form.

Syntax Rule

If a STOP RUN statement is in a consecutive sequence of imperative statements in a sentence, it must be the last statement in the sentence.

General Rules

1. STOP RUN ends image execution.
2. STOP disp suspends the image. It displays the value of disp on the user's standard display device. If the user continues the image, execution resumes with the next executable statement.

Technical Notes

1. STOP RUN causes all open files to be closed before control returns to VAX/VMS command language level.
2. STOP disp returns control to VAX/VMS command language level without terminating the image.

   The user can continue image execution with a:

   - CONTINUE command, which returns control to the program at the next executable statement
   - DEBUG command, which resumes image execution under the control of the VAX-11 Symbolic Debugger

Additional References

VAX-11 COBOL User's Guide	Debugging Programs
VAX/VMS Command Language User's Guide	

5.40 STRING Statement

Function

The STRING statement concatenates the partial or complete contents of two or more data items into a single data item.

General Format

```
STRING { | src-string |... DELIMITED BY { delim } } ...
                                        { SIZE  }
       INTO dest-string [ WITH POINTER pointr ]
       [ ON OVERFLOW stment [ END-STRING ] ]
```

src-string
is a nonnumeric literal or identifier of a DISPLAY data item. It is the sending area.

delim
is a nonnumeric literal or the identifier of a DISPLAY data item. It is the delimiter of src-string.

dest-string
is the identifier of a DISPLAY data item. It cannot be reference-modified. Dest-string is the receiving area that contains the result of the concatenated src-strings.

pointr
is an elementary numeric integer data item. It points to the position in dest-string to contain the next character moved.

stment
is an imperative statement.

Syntax Rules

1. Literals can be any figurative constant other than "ALL literal."
2. The description of dest-string cannot: (1) have a JUSTIFIED clause or (2) indicate an edited data item.
3. The size of pointr must allow it to contain a value one greater than the size of dest-string.

General Rules

1. Delim specifies the character(s) to delimit the move.
2. If the size of delim is zero characters, it never matches a src-string delimiter.
3. If src-string is a variable-length item, SIZE refers to the number of characters currently defined for it.
4. When src-string or delim is a figurative constant, its size is one character.
5. The STRING statement moves characters from src-string to dest-string according to the rules for alphanumeric to alphanumeric moves. However, no space-filling occurs.
6. When the DELIMITED phrase contains delim:
   a. The contents of each src-string are moved to dest-string in the sequence they appear in the statement.
   b. Data movement begins with the leftmost character and continues to the right, character by character.
   c. Data movement ends when the STRING operation either:
      - Reaches the end of src-string
      - Reaches the end of dest-string
      - Detects the characters specified by delim
7. No data movement occurs if the size of src-string is zero characters.
8. When the DELIMITED phrase contains the SIZE phrase:
   a. The entire contents of each src-string are moved to dest-string in the sequence they appear in the statement.
   b. Data movement begins with the leftmost character and continues to the right, character by character.
   c. Data movement ends when the STRING operation either:
      - Has transferred all data in each src-string
      - Reaches the end of dest-string.
   d. If src-string is a variable-length data item, the STRING statement moves the number of characters currently defined for the data item.
9. When the POINTER phrase is used, the program must set pointr to an initial value greater than zero before executing the STRING statement.

10. When there is no POINTER phrase, the STRING statement operates as if pointr were set to an initial value of 1.

11. When the STRING statement transfers characters to dest-string, the moves operate as if:

    a. The characters were moved one at a time from src-string

    b. Each character were moved to the position in dest-string indicated by pointr (if pointr does not exceed the length of dest-string)

    c. The value of pointr were increased by one before moving the next character

12. When the STRING statement ends, only those parts of dest-string referenced during statement execution change. The rest of dest-string contains the same data as before the STRING statement executed.

13. Before it moves each character to dest-string, the STRING statement tests the value of pointr. If it is less than one or greater than the number of character positions in dest-string, the STRING statement:

    a. Moves no further data to dest-string

    b. Executes the ON OVERFLOW phrase stment

    c. Transfers control to the end of the STRING statement if there is no ON OVERFLOW phrase

14. Subscripting or indexing evaluation for src-string and delim occur just before the STRING statement examines src-string for its delimiters.

15. Subscripting or indexing evaluation for pointr occurs just before STRING statement execution.

Additional References

Section 5.3	Scope of Statements
Section 5.27	MOVE Statement

Examples

The examples assume these data description entries:

```
WORKING-STORAGE SECTION.
01  TEXT-STRING              PIC X(30).
01  INPUT-MESSAGE            PIC X(60).
01  NAME-ADDRESS-RECORD.
    03  CIVIL-TITLE          PIC X(5).
    03  LAST-NAME            PIC X(10).
    03  FIRST-NAME           PIC X(10).
    03  STREET               PIC X(15).
    03  CITY                 PIC X(15).
```

(continued on next page)

STRING
(Continued)

```
* Assume CITY ends with "/"
    03  STATE                    PIC XX.
    03  ZIP                      PIC 9(5).

01  PTR                           PIC 99.
01  HOLD-PTR                      PIC 99.
01  LINE-COUNT                    PIC 99.
```

1. Using both delimiters and SIZE.

```
DISPLAY " ".
DISPLAY NAME-ADDRESS-RECORD.
MOVE SPACES TO TEXT-STRING.
STRING CIVIL-TITLE DELIMITED BY " "
  " " DELIMITED BY SIZE
  FIRST-NAME DELIMITED BY " "
  " " DELIMITED BY SIZE
  LAST-NAME DELIMITED BY SIZE
    INTO TEXT-STRING.
DISPLAY TEXT-STRING.
DISPLAY STREET.
MOVE SPACES TO TEXT-STRING.
STRING CITY DELIMITED BY "/"
  ", " DELIMITED BY SIZE
  STATE DELIMITED BY SIZE
  " " DELIMITED BY SIZE
  ZIP DELIMITED BY SIZE
    INTO TEXT-STRING.
DISPLAY TEXT-STRING.
```

Results:

```
Mr. Smith     Irwin  603 Main St.    Merrimack/     NH03054
Mr. Irwin Smith
603 Main St.
Merrimack, NH 03054

Miss Lambert   Alice  1229 Exeter St.Boston/        MA03102
Miss Alice Lambert
1229 Exeter St.
Boston, MA 03102

Mrs. Gilbert   Rose   8 State Street New York/      NY10002
Mrs. Rose Gilbert
8 State Street
New York, NY 10002

Mr. Cowherd    Owen   1064 A St.      Washington/    DC20002
Mr. Owen Cowherd
1064 A St.
Washington, DC 20002
```

2. Using the POINTER phrase.

```
    MOVE 0 TO LINE-COUNT.
    MOVE 1 TO PTR.
GET-WORD.
    IF LINE-COUNT NOT < 4
      DISPLAY "    " TEXT-STRING
      GO TO GOT-WORDS.
    ACCEPT INPUT-MESSAGE.
    DISPLAY INPUT-MESSAGE.
SAME-WORD.
    MOVE PTR TO HOLD-PTR.
    STRING INPUT-MESSAGE DELIMITED BY SPACE
      ", " DELIMITED BY SIZE
      INTO TEXT-STRING
      WITH POINTER PTR
      ON OVERFLOW
        STRING "                    " DELIMITED BY SIZE
          INTO TEXT-STRING
          WITH POINTER HOLD-PTR
      DISPLAY "    " TEXT-STRING
      MOVE SPACES TO TEXT-STRING
      ADD 1 TO LINE-COUNT
      MOVE 1 TO PTR
      GO TO SAME-WORD.
    GO TO GET-WORD.
GOT-WORDS.
    EXIT.
```

Results:

```
This
example
demonstrates
how
   This, example, demonstrates,
the
STRING
statement
can
   how, the, STRING, statement,
construct
text
strings
   can, construct, text,
using
the
POINTER
phrase
   strings, using, the, POINTER,
   phrase,
```

5.42 UNSTRING Statement

Function

The UNSTRING statement separates contiguous data in a sending field and stores it in multiple receiving fields.

General Format

```
UNSTRING src-string
  [ DELIMITED BY [ ALL ] delim [ OR [ ALL ] delim ] ... ]
  INTO { dest-string [ DELIMITER IN delim-dest ] [ COUNT IN countr ] } ...
  [ WITH POINTER pointr ]
  [ TALLYING IN tally-ctr ]
  [ ON OVERFLOW stment [ END-UNSTRING ] ]
```

src-string
is the identifier of an alphanumeric class data item. It cannot be reference-modified. This field is the sending area.

delim
is a nonnumeric literal or the identifier of an alphanumeric data item. It is the delimiter for the UNSTRING operation.

dest-string
is the identifier of an alphanumeric, alphabetic, or numeric DISPLAY data item. It is the receiving area for the data from src-string.

delim-dest
is the identifier of an alphanumeric data item. It is the receiving area for delimiters.

countr
is the identifier of an elementary numeric integer data item. It contains the count of characters moved.

pointr
is the identifier of an elementary numeric integer data item. It points to the current character position in src-string.

(continued on next page)

UNSTRING
(Continued)

tally-ctr
is the identifier of an elementary numeric integer data item. It counts the number of dest-string fields accessed during the UNSTRING operation.

stment
is an imperative statement.

Syntax Rules

1. Literals can be any figurative constant other than "ALL literal."
2. The symbol P cannot be used in the PICTURE character-string for dest-string.
3. Pointr must be large enough to contain a value one greater than the size of src-string.
4. The DELIMITER IN and COUNT IN phrases can be used only if there is a DELIMITED BY phrase.

General Rules

1. Countr represents the number of characters in src-string isolated by the delimiters for the move to dest-string. The count does not include the delimiter characters.
2. When delim is a figurative constant, its length is one character.
3. When the ALL phrase is present:
   a. One occurrence, or two or more contiguous occurrences, of delim (whether or not they are figurative constants) are treated as only one occurrence.
   b. One occurrence of delim is moved to delim-dest when there is a DELIMITER IN phrase.
4. When any examination finds two contiguous delimiters, the current dest-string is filled with:
   a. Spaces, if its class is alphabetic or alphanumeric
   b. Zeros, if its class is numeric
5. Delim can contain any characters in the computer character set.
6. Each delim is one delimiter. When delim contains more than one character, all its characters must be in src-string, in contiguous positions and the given order, to qualify as a delimiter.

7. When the DELIMITED BY phrase contains an OR phrase, an "OR" condition exists between all occurrences of delim. Each delim is compared to src-string. If a match occurs, the character(s) in src-string is a single delimiter. No character(s) in src-string can be part of more than one delimiter.

8. Each delim applies to src-string in the order it appears in the UNSTRING statement.

9. When execution of the UNSTRING statement begins, the current receiving area is the first dest-string.

10. If there is a POINTER phrase, the string of characters in src-string is examined, beginning with the position indicated by pointr. Otherwise, examination begins with the leftmost character position.

11. If there is a DELIMITED BY phrase, examination proceeds to the right until the UNSTRING statement detects delim. (See General Rule 6.)

12. If there is no DELIMITED BY phrase, the number of characters examined equals the size of the current dest-string. However, if the sign of dest-string is defined as occupying a separate character position, UNSTRING examines one less character than the size of dest-string. If dest-string is a variable-length data item, its current size determines the number of characters examined.

13. If the UNSTRING statement reaches the end of src-string before detecting the delimiting condition, examination ends with the last character examined.

14. The characters examined (excluding delim) as just described are:

    a. Treated as an elementary alphanumeric data item

    b. Moved to the current dest-string according to the MOVE statement rules

15. When there is a DELIMITER IN phrase, the delimiter is:

    a. Treated as an elementary alphanumeric data item

    b. Moved to delim-dest according to the MOVE statement rules

    If the delimiting condition is the end of src-string, delim-dest is space-filled.

16. The COUNT IN phrase causes the UNSTRING statement to:

    a. Count the number of characters examined (excluding the delimiter)

    b. Move the count to countr according to the elementary move rules

UNSTRING
(Continued)

17. When there is a DELIMITED BY phrase, UNSTRING continues examining characters immediately to the right of the delimiter. Otherwise, examination continues with the character immediately to the right of the last one transferred.

18. After data transfer to dest-string, the next dest-string becomes the current receiving area.

19. The process described in General Rules 11 through 18 repeats until either:

    a. There are no more characters in src-string.

    b. The last dest-string has been processed.

20. The UNSTRING statement does not initialize pointr or tally-ctr. The program must set their initial values before executing the UNSTRING statement.

21. The UNSTRING statement adds one to pointr for each character it examines in src-string. When UNSTRING execution ends, pointr contains a value equal to its beginning value plus the number of characters the statement examined in src-string.

22. At the end of an UNSTRING statement with the TALLYING phrase, tally-ctr contains a value equal to its beginning value plus the number of dest-string fields the statement accessed.

23. An overflow condition can arise from either of these conditions:

    a. When the UNSTRING statement begins, the value of pointr is less than one or greater than the number of characters in src-string.

    b. During UNSTRING execution, all dest-string fields have been processed, and there are unexamined src-string characters.

24. When an overflow condition occurs, the UNSTRING operation ends. If there is an ON OVERFLOW phrase, stment executes. Otherwise, control passes to the end of the UNSTRING statement.

25. Subscripting or indexing evaluation for src-string, pointr, and tally-ctr occur only once, just before the statement transfers any data.

26. Subscripting or indexing evaluation for delim occurs only once, just before the statement examines src-string for its set of delimiters.

27. Subscripting or indexing evaluation for dest-string, delim-dest, and countr occur just before the statement transfers data to any of these data items.

28. If there is a DELIMITED BY phrase and the size of dest-string is zero characters, no characters are moved. However, delim-dest contains the matched delimiter, and countr contains the character count.

29. If there is no DELIMITED BY phrase and the size of dest-string is zero characters, no characters are moved. The value of pointr does not change. UNSTRING continues with the next dest-string.

30. If the size of delim is zero characters, delim does not match any characters in src-string.

Additional References

Section 5.3	Scope of Statements
Section 5.27	MOVE Statement

Examples

The examples assume these data descriptions:

```
WORKING-STORAGE SECTION.
01 INMESSAGE PIC X(20).
01 THEDATE.
   03  THEYEAR  PIC XX JUST RIGHT.
   03  THEMONTH PIC XX JUST RIGHT.
   03  THEDAY   PIC XX JUST RIGHT.
01 HOLD-DELIM   PIC XX.
01 PTR PIC 99.
01 FIELD-COUNT PIC 99.
01 MONTH-COUNT PIC 99.
01 DAY-COUNT   PIC 99.
01 YEAR-COUNT  PIC 99.
```

1. With OVERFLOW phrase.

```
DISPLAY "Enter a date: " NO ADVANCING.
ACCEPT INMESSAGE.
UNSTRING INMESSAGE
  DELIMITED BY "-" OR "/" OR ALL " "
         INTO THEMONTH DELIMITER IN HOLD-DELIM
         THEDAY DELIMITER IN HOLD-DELIM
              THEYEAR DELIMITER IN HOLD-DELIM
  ON OVERFLOW MOVE ALL "0" TO THEDATE.
INSPECT THEDATE REPLACING ALL " " BY "0".
DISPLAY THEDATE.
```

Results:

```
Enter a date: 6/13/80
800613

Enter a date: 6-13-80
800613

Enter a date: 6-13 80
800613

Enter a date: 6/13/80/2
000000

Enter a date: 1-2-3
030102
```

UNSTRING
(Continued)

2. POINTER and TALLYING phrases.

```
    DISPLAY "Enter two dates in a row: " NO ADVANCING.
    ACCEPT INMESSAGE.
    MOVE 1 TO PTR.
    PERFORM DISPLAY-TWO 2 TIMES.
    GO TO DISPLAYED-TWO.
DISPLAY-TWO.
    MOVE SPACES TO THEDATE.
    MOVE 0 TO FIELD-COUNT.
    UNSTRING INMESSAGE
      DELIMITED BY "-" OR "/" OR ALL " "
             INTO THEMONTH DELIMITER IN HOLD-DELIM
             THEDAY DELIMITER IN HOLD-DELIM
                  THEYEAR DELIMITER IN HOLD-DELIM
        WITH POINTER PTR
        TALLYING IN FIELD-COUNT.
    INSPECT THEDATE REPLACING ALL " " BY "0".
    DISPLAY THEDATE "    " PTR "    " FIELD-COUNT.
DISPLAYED-TWO.
    EXIT.
```

Results:

```
Enter two dates in a row: 6/13/80 8/15/80
800613    09    03
800815    21    03

Enter two dates in a row: 10 15 80-1 1 81
801015    10    03
810101    21    03

Enter two dates in a row: 6/13/80-12/31/80
800613    09    03
801231    21    03

Enter two dates in a row: 6/13/80-12/31
800613    09    03
001231    21    02

Enter two dates in a row: 6/13/80/12/31/80
800613    09    03
801231    21    03
```

3. With COUNT phrase.

```
    DISPLAY "Enter two dates in a row: " NO ADVANCING.
    ACCEPT INMESSAGE.
    MOVE 1 TO PTR.
    PERFORM DISPLAY-TWO 2 TIMES.
    GO TO DISPLAYED-TWO.
```

(continued on next page)

```
DISPLAY-TWO.
   MOVE SPACES TO THEDATE.
   MOVE 0 TO FIELD-COUNT MONTH-COUNT DAY-COUNT YEAR-COUNT.
   UNSTRING INMESSAGE
      DELIMITED BY "-" OR "/" OR ALL " "
             INTO THEMONTH
             DELIMITER IN HOLD-DELIM COUNT MONTH-COUNT
             THEDAY DELIMITER IN HOLD-DELIM COUNT DAY-COUNT
                  THEYEAR
                  DELIMITER IN HOLD-DELIM COUNT YEAR-COUNT
        WITH POINTER PTR
        TALLYING IN FIELD-COUNT.
   INSPECT THEDATE REPLACING ALL " " BY "0".
   DISPLAY THEDATE "    " PTR "    " FIELD-COUNT
     " : " MONTH-COUNT "-" DAY-COUNT "-" YEAR-COUNT.
DISPLAYED-TWO.
   EXIT.
```

Results:

```
Enter two dates in a row: 6/13/80 8/15/80
800613    09    03 : 01-02-02
800815    21    03 : 01-02-02

Enter two dates in a row: 10 15 80-1 1 81
801015    10    03 : 02-02-02
810101    21    03 : 01-01-02

Enter two dates in a row: 6/13/80-12/31/80
800613    09    03 : 01-02-02
801231    21    03 : 02-02-02

Enter two dates in a row: 6/13/80-12/31
800613    09    03 : 01-02-02
001231    21    02 : 02-02-00

Enter two dates in a row: 6/13/80/12/31/80
800613    09    03 : 01-02-02
801231    21    03 : 02-02-02
```

5.43 USE Statement

Function

The USE statement specifies Declarative procedures to handle input-output errors. These procedures supplement the standard procedures in the COBOL Run-Time System and VAX-11 RMS.

General Format

```
USE AFTER STANDARD { EXCEPTION } PROCEDURE ON { file-name ... }
                   { ERROR     }              { INPUT       }
                                              { OUTPUT      }
                                              { I-O         }
                                              { EXTEND      }
```

file-name
is the name of a file connector described in a File Description entry in the Data Division. It cannot refer to a sort or merge file.

Syntax Rules

1. A USE statement can be used only in a sentence immediately after a section header in the Procedure Division Declaratives area. It must be the only statement in the sentence. The rest of the section can contain zero, one, or more paragraphs to define the Declarative procedures.
2. The USE statement itself does not execute. It defines the conditions that cause execution of the Declarative.
3. ERROR and EXCEPTION are equivalent and interchangeable.

General Rule

1. A Declarative executes automatically either:
   a. After standard input-output error processing ends
   b. When an invalid key or at end condition results from an input-output statement that has no INVALID KEY or AT END clause
2. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a non-zero value in a FILE STATUS data item. However, it does not execute if: (1) the condition is *invalid key* and there is an INVALID KEY phrase or (2) the condition is *at end*, and there is an AT END phrase.
3. A Declarative cannot refer to a nondeclarative procedure. In addition, a nondeclarative procedure cannot refer to a Declarative. However, PERFORM statements can refer to a USE statement or procedures associated with it.

4. After a Declarative executes, control returns to the next executable statement in the invoking routine, if one is defined. Otherwise, control transfers according to the rules for Explicit and Implicit Transfers of Control.

5. One input-output error cannot cause more than one USE AFTER EXCEPTION procedure to execute.

6. More than one USE AFTER EXCEPTION procedure can apply to an input-output operation when there is one procedure for file-name and another for the applicable open mode. In this case, only the procedure for file-name executes.

7. If an input-output error occurs and there is no applicable USE AFTER EXCEPTION procedure, the image terminates abnormally.

8. A program must not execute a statement in a USE AFTER EXCEPTION procedure that would cause execution of a USE AFTER EXCEPTION procedure that had been previously executed and had not yet returned control to the routine that invoked it.

Additional References

Section 5.5.2 Explicit and Implicit Control Transfers

Examples

```
PROCEDURE DIVISION.
DECLARATIVES.
FILEA-PROBLEM SECTION. USE AFTER STANDARD ERROR PROCEDURE
                       ON FILEA.
PROCA.
    IF FILEA-STATUS ...
ALL-EXTEND-PROBLEM SECTION.
    USE AFTER EXCEPTION PROCEDURE ON EXTEND.
PROCA.
    DISPLAY ...
I-O-PROBLEM SECTION. USE AFTER ERROR PROCEDURE ON I-O.
PROCA.
    DISPLAY ...
END DECLARATIVES.
```

1. If any input-output statement for FILEA results in an error, FILEA-PROBLEM executes.

2. If an error occurs because of an input-output statement for any file open in the extend mode except FILEA, ALL-EXTEND-PROBLEM executes.

3. If an error occurs because of an input-output statement for any file open in the I-O mode except FILEA, I-O-PROBLEM executes.

5.44 WRITE Statement

Function

The WRITE statement releases a logical record to an output or input-output file. It can also position lines vertically on a logical page.

General Format

Format 1

```
WRITE rec-name [ FROM src-item ]

[ { BEFORE } ADVANCING { { advance-num [ LINE ] } }
  { AFTER  }           { { [ LINES ]           } }
                       { { top-name            } }
                       { { PAGE                } } ]

[ AT { END-OF-PAGE } stment [ END-WRITE ] ]
     { EOP         }
```

Format 2

```
WRITE rec-name [ FROM src-item ]

[ INVALID KEY stment [ END-WRITE ] ]
```

rec-name
is the name of a logical record described in the Data Division File Section. It cannot be qualified. The logical record cannot be in a Sort-Merge File Description entry.

src-item
is the identifier of the data item that contains the data.

advance-num
is an integer or the identifier of an unsigned integer data item. Its value can be zero.

top-name
is a mnemonic-name equated to "C01" in the SPECIAL-NAMES paragraph of the Environment Division. It represents top-of-page and is equivalent to the PAGE phrase.

stment
is an imperative statement.

Syntax Rules

1. Format 1 must be used for sequential files.
2. Format 2 must be used for relative and indexed files.
3. If the File Description entry containing rec-name has a LINAGE clause, the WRITE statement cannot have an ADVANCING top-name phrase.
4. If there is an END-OF-PAGE phrase, the File Description entry containing rec-name must have a LINAGE clause.
5. The words END-OF-PAGE and EOP are equivalent.
6. In Format 2, there must be an INVALID KEY phrase if there is no applicable USE AFTER EXCEPTION procedure for the file.

General Rules

All Files

1. The file must be open in the output, I-O, or extend mode when the WRITE statement executes.
2. The record is no longer available in rec-name after a WRITE statement successfully executes. However, if the associated file-name is in a SAME RECORD AREA clause, the record is available in rec-name. It is also available in the record areas of other file-names in the same SAME RECORD AREA clause.
3. For mass storage files, the WRITE statement does not affect the Next Record Pointer.
4. The WRITE statement updates the value of the FILE STATUS data item for the file.
5. A file's maximum record size is set when it is created. It cannot be changed later.
6. On a mass storage device, the number of characters required to store a logical record in a file depends on file organization and record type. See Technical Notes.
7. WRITE statement execution releases a logical record to RMS.

Sequential Files

8. The successor relationship of records in a sequential file is set by the order of WRITE statement executions that create the file. The relationship does not change, except when records are added to the end of the file.

9. For a sequential file open in the extend mode, the WRITE statement adds records to the end of the file as if the file were open in the output mode. If the file has records, the first record written after execution of an OPEN statement with the EXTEND phrase is the successor of the file's last record.

10. When a program tries to write beyond a sequential file's externally defined boundaries, an exception condition exists:

    a. The contents of the record area are unaffected.

    b. The value of the FILE STATUS data item for the file indicates a boundary violation.

    c. If a USE AFTER EXCEPTION procedure applies to the file, it executes.

    d. If there is no applicable USE AFTER EXCEPTION procedure, the program terminates abnormally.

11. If the end of a reel/unit is recognized and the WRITE does not exceed the externally defined file boundaries:

    a. A reel/unit swap occurs

    b. The Current Volume Pointer points to the file's next reel/unit

12. The ADVANCING and END-OF-PAGE phrases control the vertical positioning of each line on a logical representation of a printed page. If there is no ADVANCING phrase, the default is AFTER ADVANCING 1 LINE.

    If there is an ADVANCING phrase:

    a. The WRITE statement advances the logical page the number of lines specified by the value of advance-num.

    b. The BEFORE phrase causes the WRITE to present the line before advancing the logical page.

    c. The AFTER phrase causes the WRITE to present the line after advancing the logical page.

    d. The PAGE or top-name phrase presents the line before or after (depending on the phrase) positioning the device to the next logical page.

       If the associated File Description entry has a LINAGE clause, the device is positioned to the first line that can be written on the next logical page, as described in the LINAGE clause.

       If there is no associated LINAGE clause, the device is positioned to the first line on the next logical page.

If page has no meaning for the associated device, PAGE and top-name are the same as ADVANCING 1 LINE. However, the BEFORE and AFTER phrases affect operation sequence.

13. If the end of the logical page is reached during execution of a WRITE statement with the END-OF-PAGE phrase, stment executes. The associated LINAGE clause specifies the logical end.

14. An end-of-page condition is reached when a WRITE statement with the END-OF-PAGE phrase causes printing or spacing in the page body footing area.

    The condition occurs when the WRITE causes the LINAGE-COUNTER to equal or exceed the value in the LINAGE clause FOOTING phrase. Stment then executes after execution of the WRITE.

15. An automatic page overflow condition occurs when the page body cannot fully accommodate a WRITE statement (with or without the END-OF-PAGE phrase).

    The condition occurs when WRITE statement execution would cause the LINAGE-COUNTER to exceed the number of lines in the page body specified in the LINAGE clause. When this happens, the line is presented on the logical page before or after (depending on the phrase) device positioning. The device is positioned to the first line that can be written on the next logical page (as described in the LINAGE clause). After execution of the WRITE, stment executes.

16. If there is no LINAGE clause FOOTING phrase, the WRITE statement operates as if the FOOTING phrase value were the same as the number of lines on the logical page. That is, the end-of-page condition occurs when the WRITE causes the LINAGE-COUNTER to equal the number of lines on the logical page.

17. If there is a FOOTING phrase and a WRITE statement would cause the LINAGE-COUNTER to exceed both the number of lines in a logical page and the value in the LINAGE clause FOOTING phrase, the WRITE operates as if there was no FOOTING phrase.

Relative Files

18. When a relative file with sequential access mode is open in the output mode, the WRITE statement releases a record to RMS. The first record has a relative record number of 1. Subsequently released records have relative record numbers of 2, 3, 4, and so on. If rec-name has an associated RELATIVE KEY data item, the WRITE places the relative record number of the released record into it.

19. When a relative file with random or dynamic access mode is open in the output mode, the program must place a value in the RELATIVE KEY data item before executing the WRITE statement. The value is the relative record number to associate with the record in rec-name. The WRITE statement releases the record to RMS.

20. When a relative file is open in the extend mode, the WRITE statement releases a record to RMS. The first record has a relative record number one greater than the highest relative record number existing in the file. Subsequent records have consecutively higher relative record numbers. If rec-name has an associated RELATIVE KEY data item, the WRITE places the relative record number of the released record into it.

21. When a relative file is open in the I-O mode and the access mode is random or dynamic, the program must place a value in the RELATIVE KEY data item before executing the WRITE statement. The value is the relative record number to associate with the record in rec-name. Executing a Format 2 WRITE statement releases the record to RMS.

22. The invalid key condition exists when either:

    a. The access mode is random or dynamic, and the RELATIVE KEY data item specifies a record that already exists in the file

    b. The WRITE tries to write a record beyond the externally defined file boundaries

23. When the invalid key condition is recognized, WRITE statement execution is unsuccessful.

    a. The contents of the current record area are not affected.

    b. The WRITE statement sets the FILE STATUS data item for the file to indicate the cause of the condition.

    c. Program execution continues according to the rules for the invalid key condition.

Indexed Files

24. Executing a Format 2 WRITE statement releases a record to RMS. The contents of the Record Keys enable later record access based on any defined key.

25. The value of the Prime Record Key must be unique in the file's records.

26. The program must set the value of the Prime Record Key data item before executing the WRITE statement.

27. If the file is open in the sequential access mode, the program must release records in ascending order of Prime Record Key values. If the file is open in the extend mode, the first released record must have a Prime Record Key value greater than the highest present in the file.

28. If the file is open in the random or dynamic access mode, the program can release records in any order.

29. When the File Description entry has an ALTERNATE RECORD KEY clause, the Alternate Record Key value can be nonunique only if there is a DUPLICATES phrase. When a program later accesses these records sequentially, the retrieval order is the same as the order in which they were written.

30. The invalid key condition is caused by any of the following:

    a. The file is open in the sequential access mode *and* in the output or extend mode, and the Prime Record Key value is not greater than the Prime Record Key value of the previous record.

    b. The file is open in the output or I-O mode, and the Prime Record Key value duplicates an existing record's Prime Record Key value.

    c. The file is open in the output, extend, or I-O mode, and the value of an Alternate Record Key for which duplicates are not allowed duplicates the value of the corresponding data item in an existing record.

    d. The WRITE tries to write a record beyond the externally defined file boundaries.

31. When the invalid key condition is recognized, WRITE statement execution is unsuccessful.

    a. The contents of the current record area are not affected.

    b. The WRITE statement sets the FILE STATUS data item for the file to indicate the cause of the condition.

    c. Program execution continues according to the rules for the invalid key condition.

If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a non-zero value in a FILE STATUS data item. However, it does not execute if: (1) the condition is *invalid key* and (2) there is an INVALID KEY phrase.

Technical Note

WRITE statement execution can result in these FILE STATUS data item values:

FILE STATUS	File Organization	Access Method	Meaning
00	All	All	Successful
02	Ind	All	Created duplicate Alternate Key
21	Ind	Seq	Attempted non-ascending key value (invalid key)
22	Ind, Rel	All	Duplicate key (invalid key)
24	Ind, Rel	All	Boundary violation (invalid key)
34	Seq	Seq	Boundary violation
92	Ind, Rel	All	Record locked by another program
94	All	All	File not open or incompatible open mode
30	All	All	All other permanent errors

Additional References

Section 5.3 Scope of Statements
Section 5.8.9 I-O Status
Section 5.8.10 Invalid Key Condition
Section 5.8.12 FROM Option
Section 5.29 OPEN Statement

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

CURSO: LENGUAJE COBOL ENFOCADO A LA
MAQUINA VAX -11
DEL 2 AL 6 DE DICIEMBRE DE 1985
DIRIGIDO AL PERSONAL PROFESIONAL DE
DIRECCION. GRAL. DE DESAROLLO TECNOLOGICO
S.C.T
MEXICO. D.F

OPCIONES DE LA

ENVIRONMENT DIVISION

DICIEMBRE DE 1985.

Opciones de

Form Management System (FMS)

Chapter 1
Introduction to VAX–11 FMS

1.1 Overview

FMS is DIGITAL's Form Management System. FMS software contains the tools for developing form applications and running them on VT100 terminals. Printed forms have been the most common tool for collecting and transmitting data in an orderly manner. FMS software now brings the speed, convenience, accuracy and low cost of computerized processing to users who have been using printed forms.

FMS was previously available only on RT-11, RSX-11M and RSX-11M-PLUS systems. In addition, many FMS application programs developed for RSX-11M or RSX-11M-PLUS systems could be run on RSX-11S. The FMS software described in this manual is designed to run on VAX/VMS V2.0.

Forms are designed by typing them directly onto the terminal screen. Neither layout charts nor a special forms design language is required. FMS associates constant data with the form, not with the application program, resulting in simplified application program maintenance and increased application program flexibility. Forms can later be modified without the need to recompile the application program.

Form application programs can be written in any language that runs on VAX/VMS. FMS provides language support for VAX-11 BASIC, VAX-11 COBOL, VAX-11 FORTRAN, and VAX-11 PL/I.

FMS software has three main components for developing and executing form application programs:

- The Form Editor (FED)
- The Form Utility (FUT)
- The Form Driver (FDV)

1.1.1 The Form Editor

The Form Editor (FED) simplifies designing, modifying, and storing form descriptions for video display. Your screen always shows the current state of the form you are working on. Keypad and keyboard functions provide ways for you to specify video display characteristics for constant text or fields that contain picture characters. To help operators, you can include in the form descriptions short, helpful explanations about individual fields and about each form as a whole.

When designing forms, you assign form names, field names, and refer to data that will be used (but not displayed) by the Form Driver when the form is used by an application. The desired operator response to information displayed or data to be entered on forms is controlled by the actual design of the form and the specific application requirements. Chapter 2 of this manual describes the Form Editor in detail.

1.1.2 The Form Utility

The Form Utility (FUT) allows you to create versions of form descriptions that are suitable for hard-copy listings, to create and modify form libraries, and to list the names of forms contained in a form library. The Form Utility also generates COBOL data division code suitable for copying into a COBOL program to correspond to a form definition. Chapter 3 of this manual describes the Form Utility in detail.

1.1.3 The Form Driver

The Form Driver (FDV) is a set of subroutines that permits your application program to access forms that you created using the Form Editor. Application programs access forms by issuing Form Driver calls that are imbedded in the program and are written in the source language of the program. All Form Driver calls refer to specific forms and/or fields within forms by names that you assign during the form editing process. The Form Driver performs field and character validation for operator input based on the form definition (validation is based on picture validation characters and field attributes). The Form Driver also responds to operator HELP requests by displaying help text associated with the form and field being processed. Chapters 4 through 7 of this manual describe the Form Driver in detail.

1.2 Developing Form Applications

The typical development cycle for form application programs has seven stages:

- PLAN

  Study the existing process that the VAX-11 FMS application will improve; list the data that operators can provide; list the hardware resources that operator sites will have; describe the skills that operators have and the additional skills they will need; specify the features that FMS forms for the application are to have and the processes that the form application programs are to perform.

- DESIGN FORMS

  Use the Form Editor to lay out and modify the forms that the form application programs will use; use the Form Utility to print form descriptions for reference, to store forms in a form library file, and to list the names of forms that are in a form library file.

- WRITE PROGRAMS

  Use the Form Driver calls in the form application to process form descriptions, to handle form-related terminal I/O, and to check (to a limited extent) the validity of operator responses.

- DEBUG PROGRAMS WITH FORMS

  Confirm that all processes that use the application's forms work.

- VALIDATE ON OPERATOR SITE SYSTEMS

  Confirm that the forms and application program software work on each type of target system on which it will be used.

- PREPARE APPLICATION SYSTEM DOCUMENTATION

  Provide complete documentation for operators who will use the FMS forms and application program software.

- DISTRIBUTE

  Package and distribute the FMS forms, application program software, and user documentation as a complete application system package.

The two major stages required when developing form applications are designing forms and writing application programs. Chapter 2 contains the information that you will need in order to design and modify forms. After forms have been designed, the Form Utility helps you to create and maintain form library files; the Form Utility is described in detail in Chapter 3.

The application program writing stage deals with the use of the Form Driver. Details for writing Form Driver application programs are contained in Chapters 4 through 8. These chapters include information on Form Driver interaction between forms and the operator as controlled by Form Driver calls issued by the application program, application programming requirements and concepts, Form Driver interface to various programming languages, the Form Driver calls, programming techniques and examples, and building and running form application programs.

Chapter 2
The VAX-11 FMS Form Editor (FED)

2.1 Overview

The FMS Form Editor allows you to create, modify, and store customized forms. Your application programs can then use these forms to collect data entered by an operator at a video terminal.

Creating or editing a form with the Form Editor is an interactive and iterative process. You do not need to know all the details or all the modifications that you intend to specify for a form. The Form Editor lets you test various possibilities, observe their appearance on the screen, and choose the design that you consider most successful.

The product of your work with the Form Editor is a form description that can be saved in a file or form library. The form description can be retrieved from the file or form library for additions or changes. You can change individual fields or text portions of the form without affecting other fields or text.

For example, you might want to translate the text of a form into another language, reposition items on the screen to make the form more attractive to the eye or easier for an operator to handle, add or remove fields, or supply additional help text. You can make these and other changes by calling the Form Editor, editing the screen image of the form to make the desired changes in the form description, and saving the modified form description in a form description file.

The purpose of the form description is to provide information to another software component called the Form Driver. The Form Driver handles the interaction of the terminal operator with the form displayed on the screen and with the application program. The Form Driver is described in Chapter 4.

In summary, the Form Editor allows you to perform these operations:

1. Creation and modification of a form's screen image by means of the terminal's main keyboard and the text editor keypad.
2. Storage of form descriptions in files, and retrieval of the form description files from form libraries.

• Issuing Form Editor Commands

You can use any of several commands to enter a particular phase of the Form Editor. Table 2-1 summarizes the commands. You type the commands in response to the COMMAND: prompt. If you type HELP in response to the COMMAND: prompt, the Form Editor displays the valid responses to the prompt.

Table 2-1: FED Command Summary

Command	Abbreviated Command	Function
EDIT	ED	Create or edit the form's screen image.
ASSIGN *[option]*	A	Assign field attributes. ("Attributes" are characteristics of fields that you assign with FED for use by the Form Driver.)
where *option* can be one of the following:		
ALL	A A	Assign attributes for old and new fields.
NEW	A N	Assign attributes for newly created fields.
FIELD *fldnam*	A F *fldnam*	Assign attributes for the field called *fldnam*.
FORM	F	Assign form wide attributes. (Form wide attributes are attributes that apply to an entire form rather than a particular field.)
HELP	H	Lists the commands available in the form editor.
NAME	N	Enter and edit named data. (Named data is information that is to be associated with a form but not displayed with it.)
SAVE	None	Store the form and return to the FED> prompt. Both input and output files are preserved.
QUIT	None	Cancel a session without saving output files and return to the FED> prompt. The input file is preserved.

• File Specifications

The Form Editor's output always goes to a form file. A form file contains one form description. (To create or update form libraries use the Form Utility (FUT), described in Chapter 3.)

To create a new form, use the /CR option:

```
FED>/CR↵
```

To edit a form contained in a form file, type the name of the form file, for this example "VENDOR".

```
FED>VENDOR↵
```

To extract a form from a form library for editing, type the name of the library file and respond to FED's Form name? prompt with the name of the form. The default file type is .FRM, indicating a form file. If the file type is .FLB, you must type .FLB:

```
FED>DEMLIB.FLB↵
Form name? FIRST↵
```

If the specified form is not found, FED repeats the Form name? prompt. If you press the ↵ key in response to the Form name? prompt, FED displays the FED> prompt again and waits for a new command line.

It is not necessary to distinguish a form file from a library file on the command line. FED determines whether the input file is a form file or a library file, and proceeds accordingly.

If you want to know the version number of the Form Editor you are using, you can ask the Form Editor to display its identification message by typing:

```
FED>/ID↵
```

The default file type for the input file is FRM (a form file). If an explicit version number is not specified for an input file, FED uses the latest version of the file.

The output file that FED creates during an editing session is always a form file with the file specification "*form*.FRM". "*Form*" is the name of the form when the session ends. FED creates the output file in the current default directory.

2.4 Form Editor Commands

You can type any one of the commands shown in Table 2-1 (The FED Command Summary) in response to the COMMAND: prompt. The Form Editor enters the specified command *after* you press the ENTER key on the keypad. If you want to cancel your last command, type the ^Z↵ combination before

2.2 Form Editor Terminology

2.2.1 Screen Form

The screen form looks like a paper form but is instead a video display. The computer displays a form by using a form description that specifies to the computer which characters to display on the screen.

2.2.2 Form Description

The form description is the computer's specifications of a screen form. It specifies which characters to display on the screen as well as the location, size and other characteristics of each field. The name of the form and how the form and its fields are processed are also part of the specifications.

2.2.3 Field

The field is a set of contiguous characters (either picture-validation or field-marker) terminated by a blank, a non-field character, an end-of-line delimiter or a change in video attributes. A field is a formatted blank for some of the information that a form has been designed to work with.

2.2.4 Form Description File

The form description file is a computer file that contains only one form description which may or may not be complete or accurate. It is a binary file that has been arranged so FMS can use it to display a screen form.

2.2.5 Form Library File

The form library file is a computer file containing at least one form description and a directory of the names for each form description. It is a binary file but is arranged so individual form descriptions can be accessed by name for use by FMS.

2.3 Starting the Form Editor

The Form Editor (FED) requires a VT100 terminal. The terminal must be made known to the system as a VT100. Use the SET command for this:

```
$ SET TERMINAL/VT100 ↵
```

(See the *VAX/VMS User's Guide* for a description of the SET command.)

The main keyboard performs normally when you are using the Form Editor, providing you with a means to insert characters, delete them, and so on. The keypad to the right of the keyboard provides operations specifically related to the Form Editor.

The following operations are used to design a form:

• Starting the Form Editor

By using the standard commands to load the Form Editor into memory, you begin program execution.

When the Form Editor prompt (FED>) is displayed on the screen, you may type in a response. The response describes the form file that you want to create or edit, or the library that contains the desired form. The response requires a prescribed syntax (described in the "File Specification" section).

To start the Form Editor type:

```
$ MCR FED ↵
```

You can create a symbol so you do not have to re-type the command string every time you want to use the Form Editor.

```
$ FED:==MCR FED ↵
```

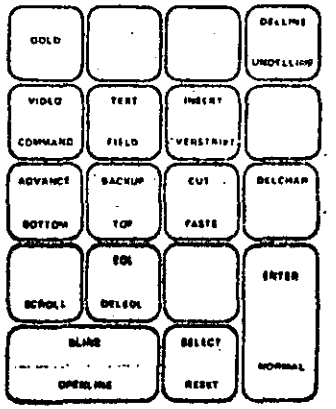
The Form Editor clears the screen, displays the prompt FED> at the bottom of the screen and accepts a command line.

The Form Editor is built with buffer space that is sufficient to edit any form.

• Using Keypad Operations

The keypad layout for the Form Editor in Figure 2-1 shows the operations that are associated with certain keys or key combinations.

Figure 2-1: FED Keypad Layout



you press the ENTER key. If you want to change a command, use the DEL key to delete the characters that make up the command. You can use CTRL/U to delete the entire command line and then type in a new command.

You can type the HELP command to display a list of the Form Editor commands and their functions.

Begin an editing session with a rough pencil sketch of the form that you want to create. You can elaborate the details of the form interactively with the Form Editor by looping back through the command functions (by means of the GOLD/COMMAND key sequence) and adding or deleting features gradually during the development of the form design.

The various command operations let you control the phases of your work during a form editing session and to move in an orderly manner from one phase to another. Any phase can be entered at any time. The FORM, ASSIGN, and NAME phases use the Form Driver to display and collect responses with questionnaire forms.

Form Driver key operations are active while you are completing any of the questionnaires. For example, when you are assigning form wide, field, and named data attributes, the TAB key has the effect of moving the cursor to the first character position of the next field, and the BACKSPACE key moves the cursor to the previous field. Chapter 4, on Form Driver interaction with the terminal operator, describes Form Driver key operations in detail.

2.4.1 Assigning the Form Wide Attributes: The FORM Command

FORM places you in the Form Wide Attribute Questionnaire. The Form Editor displays a questionnaire that collects the necessary information from you to create a form file.

2.4.2 Editing the Form Display: The EDIT Command

EDIT causes the Form Editor to enter the EDIT phase. In EDIT you create and modify the screen image of the form. You may type background text, create fields and scrolled areas, and assign some kinds of attributes. Use the GOLD/COMMAND key sequence to return to the COMMAND: prompt.

2.4.3 Assigning the Field Attributes: The ASSIGN Commands

ASSIGN with any of its options tells the Form Editor to enter the field attribute assignment phase. When the form is a new one, you usually type ASSIGN after completing the EDIT phase. For each field in the form, the Form Editor displays a questionnaire that requests field attributes. If you are editing an existing form, you only need to fill in field attributes that were not assigned earlier.

If, during the ASSIGN phase, you wish to exit before completion of all field attribute assignments, press the period (.) key on the keypad. This action returns you to the COMMAND: prompt and assigns default attributes to all remaining fields.

2.4.3.1 For All Fields: The ASSIGN ALL Command — Causes the Form Editor to request attributes for all fields. To display the questionnaire for the next field, press the ENTER key.

2.4.3.2 For New and Changed Fields Only: The ASSIGN NEW Command — Causes the Form Editor to request attributes for new fields only. To display the questionnaire for the next field, press the ENTER key.

2.4.3.3 For a Specified Field Only: The ASSIGN Field Command — Followed by a field name, allows you to assign attributes to the field you specify.

2.4.4 Specifying the Named Data: The NAME Command

NAME places you in the named data assignment phase. The Form Editor displays an Entry Form that collects names and data to be associated with those names. Named data is typically used to hold information about a form in the form description but outside the form itself. Named data is not displayed with a form.

2.4.5 Storing the Form Description: The SAVE Command

The SAVE command stores the form description that you are working on in an output file and returns you to the FED> or system prompt, depending on how you started the program.

If field attributes have not yet been assigned to all fields when the SAVE operation is performed, the Form Editor supplies default values for any fields whose attributes have been left unspecified; a default name of all blanks is supplied as the field name.

2.4.6 Canceling the Session Without Saving the Form: The QUIT Command

QUIT returns you to the FED> or system prompt. The form you were editing is not saved in the output file; it is destroyed.

2.5 Edit Status Display

When you are in the EDIT phase, the bottom line (24) of the screen displays information about the current status of the Form Editor. The format for the line is:

```
CURSOR: TXT NOR LIN   1 COL    1 MODES: TXT ADV INS  SELECT: LIN   1 COL    1
        FLD SCR      23      132        FLD SCR OVR               23      132
```

The second line (above) indicates the alternative choice or the limitations of the items in the display.

The fields on line 24 are displayed in reverse video.

CURSOR This section indicates the line and character that the cursor is located on.

TXT,FLD The cursor character is either a text (TXT) character or a field (FLD) character.

NOR,SCR The cursor line is either a normal screen line (NOR) or a part of a scrolled region (SCR).

LIN 1-23 The line number at which the cursor is located.

COL 1-132 The column number at which the cursor is located.

MODES This section indicates the status of the internal mode indicators of the editor.

TXT,FLD The current input mode is either text or field.

ADV,BCK The current move mode is either Advance (ADV) or Backup (BCK).

INS,OVS The current input mode is either Insert (INS) or Overstrike (OVS).

SELECT This section is present only if a select range is active. Otherwise, this portion of the line is blank.

LIN 1-23 The line number at which the select point is located.

COL 1-132 The column number at which the select point is located.

2.6 Form Editor Operations Reference

This section describes the creation of the form's screen image during the EDIT phase and the assignment of all attributes during the FORM, EDIT, ASSIGN, and NAME phases.

2.6.1 Creating the Form's Screen Image

The Form Editor includes a text editor for creating and modifying screen images. The text editor lets you use standard operations for mode changing, cursor control, and text modification.

The Form Editor lets you define fields in the form for data input/output between your application and the terminal operator. It also enables you to assign video attributes (such as bold, blink, and underline) to any character or set of characters on the terminal screen, and to define a block of lines as a scrolled area.

2.6.2 The Text Editor

The keyboard performs like a typewriter when you use the Form Editor: it lets you enter and delete characters. The keypad to the right of the keyboard provides operations specifically related to the Form Editor. It is recommended that you make a copy of the keypad layout and keep it at the terminal.

The text editor provides four kinds of operations:

- **Mode-Changing Operations**

  You change modes by pressing the appropriate key or key combination on the keypad. Modes determine placement of characters, movement forward or backward through the form, and definition of fields and background text.

- **Cursor Control Operations**

  These operations change the cursor position but do not affect the text. The cursor may advance only to the margin boundaries.

- **Text Modification Operations**

  These operations insert, delete, and modify text.

- **Scroll Operation**

  This operation permits the definition of a scrolled line. Together with identical lines that immediately follow it, the line becomes a scrolled area.

2.6.3 Mode-Changing Operations

The Form Editor works in several modes. The mode choices are TEXT/FIELD, INSERT/OVERSTRIKE, and ADVANCE/BACKUP. Only one of each pair can be active at one time.

The TEXT/FIELD modes tell the Form Editor whether the characters you enter are background text characters for the form (TEXT mode), or the special set of field characters that define the picture format of a field (FIELD mode). The special set of field characters includes field-markers (such as slashes and hyphens that delimit fields) and picture-validation characters.

The INSERT/OVERSTRIKE modes determine how the Form Editor places characters in the form with respect to characters already there.

The ADVANCE/BACKUP modes determine whether the Form Editor executes an operation in a forward (right and downward) or backward (left and upward) direction.

2.6.3.1 TEXT/FIELD — Enter TEXT mode by pressing the TEXT key on the keypad. Start FIELD mode by pressing the GOLD/FIELD key sequence. In TEXT mode, the Form Editor accepts any character as input. It enters any printable character or space in the background text of the form. The Form Driver does not see these characters as data. Rather, it treats the characters as constant text that is always displayed on the form. Return to FIELD mode by pressing the GOLD/FIELD key sequence.

In FIELD mode, the Form Editor accepts as input only the picture-validation characters A, C, N, X, and the digit 9, as well as a set of ASCII field-marker characters. Picture-validation characters tell the Form Editor whether to ac-

cept alphabetic (A), alphanumeric (C), numeric (9), signed numeric (N), or any characters (X) as input for each character position in a field. Field-marker characters, such as the pound sign (#) and the dash (—), are text characters that you may define as part of a field.

If, while in FIELD mode, you enter a character that is neither a field-marker nor a picture-validation character, the Form Editor sounds the terminal bell and rejects the input. The Form Editor accepts a blank as input in FIELD mode, but does not make it part of the field. Field-marker and picture-validation characters are treated as such only when the Form Editor is explicitly in FIELD mode. For example, the digit 9 is associated with a field as a picture-validation character if it is typed in FIELD mode; otherwise, it is treated as a text character.

You can change FIELD mode to TEXT mode by pressing the TEXT key.

2.6.3.2 ADVANCE/BACKUP — The ADVANCE/BACKUP modes affect the BLINE (beginning of line) and EOL (end of line) operations. They do not affect character insertion or deletion.

ADVANCE mode causes the Form Editor to implement operations in the direction moving from the current cursor position toward the end of the line or form. You can deactivate ADVANCE mode by pressing the BACKUP key.

BACKUP mode causes the Form Editor to implement operations in the direction toward the beginning of the line or form. You can deactivate BACKUP mode by pressing the ADVANCE key.

2.6.3.3 INSERT/OVERSTRIKE — The INSERT/OVERSTRIKE modes affect the way characters are placed or moved when you type or make deletions.

INSERT mode places typed characters at the current cursor location and moves the cursor to the right. Any other characters on the line are moved over to make room for the inserted character. If characters would be lost by being pushed beyond the margin, the Form Editor sounds the terminal bell and rejects the insertion.

If you delete a character in INSERT mode, the Form Editor removes the character to the left of the cursor and characters to the right slide over to close the space.

You can deactivate INSERT mode by pressing the GOLD/OVERSTRIKE key sequence.

OVERSTRIKE mode causes the Form Editor to replace the character at the current cursor position with the new character typed at the terminal. When a character is deleted, adjacent characters do not close up the line. The character is erased. The deleted character is replaced by a blank, and the cursor is positioned on that character's space. You can enter OVERSTRIKE mode by typing the GOLD/OVERSTRIKE key sequence.

Deactivate OVERSTRIKE mode by pressing the INSERT key.

2.6.4 Cursor Control Operations

The following operations change the cursor's position during an editing session.

The cursor symbol (either a solid rectangle or an underline) blinks on the character cursor location. A row-column counter in the lower right corner of the screen displays the character position where the cursor symbol is blinking.

Uparrow (↑) Press the UPARROW key to move the cursor up one line. You cannot move the cursor above the top margin of the form, otherwise the Form Editor sounds the terminal bell.

Downarrow (↓) Press the DOWNARROW key to move the cursor down one line. You cannot move the cursor below the bottom margin of the form, otherwise the Form Editor sounds the terminal bell.

Rightarrow (→) Press the RIGHTARROW key to move the cursor one character position to the right. You cannot move the cursor beyond the right margin, otherwise the Form Editor sounds the terminal bell.

Leftarrow (←) Press the LEFTARROW key to move the cursor one character position to the left. You cannot move the cursor beyond the left margin, otherwise the Form Editor sounds the terminal bell.

BLINE Press the BLINE key to move the cursor to the beginning of a line. Which line the cursor moves to the beginning of depends on whether the Form Editor is in ADVANCE or BACKUP mode when the BLINE key is pressed.

If the Form Editor is in ADVANCE mode, BLINE moves the cursor to the beginning of the next line. Pressing BLINE again moves the cursor to the beginning of the subsequent line.

If the Form Editor is in BACKUP mode, BLINE moves the cursor to the beginning of the current line. Pressing BLINE again moves the cursor to the beginning of the previous line.

If an attempt is made to move to a line beyond the top or bottom screen boundary, the Form Editor sounds the terminal bell.

RETURN The RETURN key on the keyboard provides an alternative to LINE when used in ADVANCE mode. Pressing RETURN moves the cursor to the beginning of the next line. BACKUP mode has no effect on this operation.

EOL — Pressing the EOL key moves the cursor to the end of a line. Which line the cursor moves to the end of depends on whether the Form Editor is in ADVANCE or BACKUP mode.

If the Form Editor is in ADVANCE mode, EOL moves the cursor to the end of the current line. If you strike EOL again, the cursor moves to the end of the next line.

If the Form Editor is in BACKUP mode, EOL moves the cursor to the end of the previous line.

BOTTOM — Pressing the GOLD/BOTTOM key sequence on the keypad moves the cursor to the bottom right corner of the screen.

TOP — Pressing the GOLD/TOP key sequence on the keypad moves the cursor to the top left corner of the screen.

REPEAT — If you press the GOLD key, a number, and an operation that you want to perform, the Form Editor repeats that operation the number of times that you have specified. After you type the first digit of the number, you see the prompt REPEAT: on the screen as well as the number itself. The first command or key typed after the digits is repeated that number of times. You can edit the number using the DELETE key to increase or decrease the repetitions.

2.6.5 Text Modification Operations

Text modification operations allow you to insert, modify, and delete characters and lines in the form, as well as to assign video attributes to background text and fields

The Form Editor handles typed characters differently depending on whether INSERT or OVERSTRIKE mode is in effect.

2.6.5.1 Inserting ASCII Characters — When you type any ASCII character, the Form Editor inserts that character at the current cursor location and moves the cursor one location to the right.

If you type a character at the end of a line, the Form Editor inserts the character in the last available position, sounds the terminal bell, and causes the cursor to "bounce back," leaving the cursor symbol at the last character position on the line.

2.6.5.2 Inserting Characters in INSERT Mode — In INSERT mode, the Form Editor inserts the character at the current cursor position. The character previously located there moves one position to the right. All other characters on the line to the right of the cursor move one position to the right. If the last character on the line is not a blank, the Form Editor rejects any operation that would cause that character to be lost by pushing it off the end of the line. If any fields are moved on a line, the Form Editor automatically updates their field descriptors in the form description to reflect the change in the field's screen location.

Press the DELETE or  key on the keyboard to delete the character to the left of the cursor. If the cursor position is in column 1 when the delete key is pressed, the Form Editor rejects the operation and sounds the terminal bell.

DELETE moves the cursor and the remaining characters on the line one character position to the left. A blank is inserted at the end of the line.

2.6.5.3 Inserting Characters in the OVERSTRIKE Mode — In OVERSTRIKE mode, the Form Editor replaces the character at the current cursor position with the new character that is typed.

Press the DELETE or  key on the keyboard to delete the character to the left of the cursor. If the cursor position is in column 1 when this key is pressed, the Form Editor rejects the operation and sounds the terminal bell.

If the Form Editor is in OVERSTRIKE mode, DELETE replaces the character to the left of the cursor with a blank and moves the cursor one position to the left. If a field's position is changed, the corresponding descriptor is updated. However, if a field's picture is modified, it is a new field and old attributes are lost.

2.6.5.4 DELETE CHARACTER — Press the DEL(ete)CHAR(acter) key on the keypad to delete the character at the cursor position. If a field's position is changed, the descriptor is updated. However, if a field's picture is changed, it is, for all intents, a new field and the old attributes are lost.

If the Form Editor is in INSERT mode, DELCHAR deletes the character, moves the remaining characters on the line one position to the left , and inserts a blank at the end of the line. The cursor remains in its current position.

If the Form Editor is in OVERSTRIKE mode, DELCHAR replaces the character on which the cursor is positioned with a blank and moves the cursor one position to the right. This is equivalent to typing a blank while in OVERSTRIKE mode. If the cursor is on the last character position on the line, the Form Editor deletes the character, sounds the terminal bell, and leaves the cursor in its current position. The Form Edit or updates the field descriptors of fields affected by the change.

2.6.5.5 OPENLINE — Press the OPENLINE key to insert a blank line at the current line and move all remaining lines down one line. The Form Editor reassigns screen locations to affected fields on the form that already have field descriptors. If the next to last line on the screen (the last line available for your form) is not blank, the Form Editor rejects the OPENLINE operation, sounds the terminal bell, and prints an error message.

2.6.5.6 — Press the combination to redisplay the current screen, and restore the keypad to application mode. This command is useful when there are power failures, static problems, or distortions.

2.6.5.7 (GOLD) (U) — Press the (GOLD) (U) combination to delete all characters between the current cursor position and the beginning of the line. The cursor remains at its current position.

2.6.5.8 DELEOL — Press the DELEOL (DELete End Of Line) key on the keypad to delete all characters between the cursor location and the end of the line, replacing them with blanks. The cursor remains at its current position.

2.6.5.9 DELLINE — Press the DEL(ete) LINE key to delete the current line, move all the lines below it up one line, and insert a blank line at the bottom. The Form Editor updates the field descriptors of any affected fields. The UNDELLINE operation allows you to recover the deleted line. The entire line is deleted regardless of the cursor position in the line.

2.6.5.10 UNDELLINE — Press the UNDEL(ete) LINE key to restore the line or line segment that you have just deleted. This operation saves you from mistaken or accidental deletions. It also provides you with an easy way to duplicate lines. For example, UNDELLINE can be used to create many identical lines in a scrolled area.

The effect of the UNDELLINE operation depends on how the original deletion was performed.

If the deletion was performed by using a DELEOL or a (GOLD) (U), the Form Editor places the contents of the buffer containing the deleted characters at a position starting at the current cursor location. If deleted by (GOLD) (U), the characters are placed to the left of the cursor location; if by DELEOL, they are placed to the right of the cursor location. This restoration can be performed only if the deleted characters will be replacing blanks. Field descriptors for the original fields are restored only if the cursor remains at the location where the original deletion was made.

If you made the deletion with DELLINE, the Form Editor performs an OPENLINE operation at the current cursor position. It then places the deleted line on the screen in the blank line created by OPENLINE. The Form Editor updates all old field descriptors for fields affected by the OPENLINE operation when the field's position changes, but not the picture. The field descriptors for the deleted line are restored only when UNDELLINE is performed the first time and on the same line where the deletion was done.

2.6.5.11 REPEAT — To repeat a character, or an operation, press the GOLD key, and type a number. The Form Editor repeats that operation the number of times that you have specified. After you type the first digit of the number, the prompt REPEAT: and the number appear on the screen. The first command or key typed after the digits is repeated that number of times. You can edit the number to increase or decrease the repetitions by using the (BS) and (GOLD) (U) operations. (BS) is not a repeatable function.

2.6.5.12 SELECT — Press the SELECT key to mark the current cursor position as a reference point for video attribute assignment and CUT operations. SELECT defines the first character of the select range. The end of the select range is the final position to which you move the cursor. In other words, the select range is defined as all character positions in the area delimited by the SELECT position at one corner and the current cursor position at the other. SELECT is used with the CUT, PASTE and VIDEO operations.

2.6.5.13 CUT — Pressing the CUT key saves all the characters contained in the current select range (the area defined by the place where SELECT was pressed and the current cursor position). The characters are stored in a buffer, and blanks replace the contents of the area in the screen image. If a SELECT operation has not been performed, the Form Editor sounds the terminal bell in response to an attempted CUT.

2.6.5.14 PASTE — The PASTE operation inserts the characters saved by CUT into the same area relative to the current location of the cursor as obtained when the original CUT operation occurred. The PASTE operation checks that the inserted material does not cross boundary lines or any other text or fields in the form. If boundary lines are crossed, the Form Editor displays the error message "Cannot paste over margins or non-blanks or in scrolled areas" and sounds the terminal bell.

The PASTE operation is allowed only if the target paste area consists entirely of blanks. If the target paste area is not blank, the target area is painted in reverse video, and a message is displayed on line 24. When this occurs, press any key to remove the reverse video attribute, move the cursor to define a proper target area, and continue the operation.

2.6.5.15 VIDEO — Press the VIDEO key to assign video attributes to the form. The prompt VIDEO: appears on the terminal screen. Type any of the following responses to activate the specified attribute within the select range. Press the ENTER key after typing the response. The abbreviations are underlined:

Bold	Displays all characters within the select range in bold face.
Blink	Displays all characters within the select range in alternately increasing and decreasing screen brightness.
Reverse	Displays all characters within the select range on a reverse screen background. If the screen is white-on-black, characters in reverse video appear in black-on-white; if the screen is black-on-white, the characters appear in white-on-black.
Underline	Underlines all characters within the select range.
Clear	Deactivates or clears all the active video attributes in the select range.
Edit	This is not an attribute, but returns you to the normal screen editing mode.

You must use the SELECT operation (see above) to delimit the characters affected. The SELECT range includes both text and fields; it may cut a field in the middle. A field cut in two by the SELECT operation becomes two separate fields if the two parts of the field receive different video attributes.

You can assign video attributes in either TEXT or FIELD mode.

Since you can use the CLEAR attribute to cancel the other video attributes, you can easily experiment with the various attributes to achieve the best effect. When you have the combination of attributes that you want to keep in your form, end the video attribute assignment session by typing EDIT or pressing the RETURN key.

A character can have more than one video attribute. For example, the character can appear on the operator's screen as both bold and blinking. However, all characters in a field must have the same video attributes.

2.6.6 Scroll Operation

By using the scroll function in the EDIT phase of the Form Editor, you can set up scrolled areas in a form. The Form Driver can scroll lines in response to subroutine calls from your program (see Chapter 7 for a scrolling example).

The scrolled area that the Form Editor and Form Driver work with is like a "window" into a collection of data too large to appear on the screen at one time. Your program must store and manipulate any data that scrolls off the screen. The Form Driver does not have the capacity to store such data.

Scrolling, in effect, allows you to create a form of unlimited length that can be filled in by an operator as information becomes available. An inventory clerk receiving lists of needed materials continuously during the day, or a bank teller recording ongoing transactions, could use a scrolled area in a form application.

Pressing the GOLD/SCROLL key sequence tells the Form Editor to define the current line as scrolled. A scrolled area is a minimum of two lines.

The GOLD/NORMAL key sequence removes the scrolling attribute from a line.

Once you have defined a line as scrolled, you can extend the scroll and create a scrolled area by using the DELLINE and UNDELLINE operations. Delete the scrolled line and then "undelete" or restore it as many times as you wish. In this way, you can be sure that the lines of the scrolled area are identical.

The GOLD/SCROLL key sequence only defines the current line as scrolled. The succeeding lines that are identical to the scrolled line are processed as part of the scrolled area. The first line that differs in any detail from the original scrolled line causes the Form Editor to terminate the scrolled area.

All lines in a scrolled area must have identical fields. A scrolled area should not contain text except for field-marker characters. Once the text scrolls off the screen, it is lost.

The Form Editor, during its field attribute assignment phase, asks you about the fields on the first line of a scrolled area only. Fields on subsequent lines of the scroll are considered to have the same attributes as the fields on the first line. A form may have more than one scrolled area.

2.6.7 Field Pictures

A field is a set of contiguous field characters (picture-validation or field-marker characters) terminated by a blank, a non-field character, an end-of-line delimiter, or a change in video attributes. Picture-validation attributes apply only to characters in fields. They tell the Form Driver whether the operator may enter a number, a letter, etc., in response to a given field.

The Form Editor recognizes the five picture-validation characters shown in Table 2-2.

Table 2-2: FED Picture-Validation Characters

Character	Type
C	Alphanumeric
A	Alphabetic
9	Numeric
N	Signed Numeric
X	Any Character

2.6.7.1 For Alphanumeric Characters—C — The C in any character position defines what is valid input in that position. The C character is a character attribute allowing the operator to enter the digits 0 through 9, the letters A through Z (either in upper or lower case) and/or a space. Any other attempted input sounds the terminal bell and causes an error message.

2.6.7.2 For Letters—A — The A in a character attribute position indicates to the operator to enter the letters A through Z (either in upper or lower case) and a space.

2.6.7.3 For Unsigned Numbers—9 — The 9 in a character attribute position indicates to the operator to input only the digits 0 through 9.

2.6.7.4 For Signed Numbers—N — The N in a character attribute position allows the operator to input the digits 0 through 9, with only one decimal point and with only one plus (+) sign or one minus (-) sign. Their positions within the field are not checked by the Form Driver. Any other input is rejected.

2.6.7.5 For Any Printable Characters—X — The X in a character attribute position allows the operator to input any displayable character.

2.6.7.6 For Mixed Pictures — A single field may contain different picture-validation characters. For example, a field constructed to accept both alphabetic and numeric characters specifically may look like this:

```
AAA999
```

Such a field allows the operator to enter alphabetic characters in the first three field character positions and digits in the last three field character positions. The field is said to have a "mixed picture."

2.6.7.7 With Field-Marker Characters — For example, a field whose picture looks like this

```
999#AA-99
```

contains two field-marker characters, the pound sign and the dash.

The Form Editor treats all field-marker characters — whether leading, trailing, or embedded — as part of the field in which they occur.

A field that contains field-marker characters but only one picture-validation character does not have a mixed picture. Two or more picture-validation characters in a single field constitute a mixed picture.

Field-marker characters may be the ASCII characters from 41 to 57 octal and 72 to 100 octal (Table 2-3). The Form Editor accepts field-marker characters when in FIELD mode. The Form Driver does not return field-marker characters to the calling program or include them in the length of the field. Field-marker characters are transparent to the program, which does not pass them to the Form Driver in the data to be displayed in a field.

Table 2-3: FED Field-Marker Characters

Character	Character
!	.
"	.
#	/
$	
%	:
&	;
'	<
	=
(	>
)	?
*	
+	@

2.6.8 Assigning Form Wide Attributes

The Form Editor collects Form Wide Attributes by displaying the questionnaire shown in Figure 2-2. The Form Editor automatically displays the Form Wide Attributes questionnaire when you create a new form or when you type FORM in response to the COMMAND: prompt. The questionnaire contains the default conditions for each choice.

Figure 2-2: Form Wide Attributes Questionnaire

```
                    Form Wide Attributes

Form Name       : [      ]
Help Form Name  : [      ]
Reverse Screen (Y,N)
Current Screen (Y,N)
Wide Screen    (Y,N)
Starting Line (1,23)
Ending Line   (1,23)

Impure Area   ???? bytes
Form Size     ???? words
```

MR-041-80

Press the TAB key to move from one question to the next. You can use the BACKSPACE key to move backward. When you have completed the input and want to exit, return to the COMMAND: prompt by pressing the ENTER key.

The fields listed in the Form Wide Attributes Questionnaire are:

2.6.8.1 Form Name — A response to this field is required. When FED saves the form description in a file, the file name is the form name with a file type of .FRM. When you use FUT to place the form description file in a form library, the form name is used as the form description file name.

2.6.8.2 Help Form Name — This field contains the name of an associated Help form. The field may be left blank.

2.6.8.3 Reverse Screen — If this field contains a Y, the form is displayed black-on-white. If it contains an N, the display is white-on-black. The default is N.

2.6.8.4 Current Screen — If this field contains a Y, the Form Driver displays the form in the current screen mode. An 80-column form with a Y answer to this field does not require a change if the current mode is set at 132 columns.

The current screen also applies to reverse screen. If current screen is specified, the Form Driver does not change the screen background or the screen width, unless the form is specified for 132 columns and the screen is currently 80 columns.

You cannot specify Y both to this option and to the wide screen option described below. If the choice is N, the Form Driver resets the screen if necessary to conform to the display mode for this form. The default value is N.

2.6.8.5 Wide Screen — If this field contains a Y, the form is displayed in 132-column mode and the Current Screen option described above is set to N. If the field contains an N, the form is displayed in 80-column mode. The Form Editor changes the terminal to the selected mode. The default is N.

2.6.8.6 Starting Line — This field contains a value from 1 to 23 inclusive, indicating the first line of the screen to be cleared when the form is displayed. If you specify a starting line number greater than the ending line number, the Form Editor replaces your entry with the default value of 1.

2.6.8.7 Ending Line — This field contains a value from 1 to 23 inclusive, indicating the last line of the screen to be cleared when the form is displayed. If the value is less than the starting line number, the default value of 23 is used by the Form Editor.

Starting and ending line number defines the area of the screen to be cleared when the form is displayed using the FDV$SHOW call (which doesn't automatically clear the entire screen) or when the form is displayed as a help form.

2.6.8.8 Impure Area — This is a Display-Only field that indicates the size of the impure area required when the form is displayed by the Form Driver. You specify the array and size of the impure area in the FDV$INIT call within your program. The two arguments passed are: the array, which must be at least 8 (32 bit) integers; and the size, which must be 32 bytes (8 longwords). When creating a new form, this field is initially displayed as question marks.

2.6.8.9 Form Size — This is a Display-Only field that indicates the length of the form. This value is used in calculating the media or memory storage requirements for the form. When creating a new form, the field is initially displayed as question marks until the Form Editor determines the correct value.

2.6.9 Assigning Field Attributes

The Form Editor collects field attributes by displaying the questionnaire shown in Figure 2-3. Each entry in the questionnaire designates a single attribute for a field. If the form or field is a new one, the Form Editor supplies default values in the questionnaire. If the attributes for the field were assigned in a previous editing session, those values are displayed.

Figure 2-3: Field Attributes Questionnaire

```
Name    : FIELD1  Right Just (Y,N) N  Clear Char (chr)    Zero Fill (Y,N) N
Default :
Help    : Type_a_1,_2_or_3.
Auto Tab   (Y,N) N  Resp Reqd (Y,N) N  Must Fill (Y,N) N  Fixed Dec (Y,N) N
Indexed (N,U,Y) N  Disp Only (Y,N) N  Echo Off  (Y,N) N  Supv Only (Y,N) N
```

MR-043-80

Enter the attribute assignment phase by typing ASSIGN and any of the following options in response to the COMMAND: prompt.

ASSIGN NEW	Assign attributes only to new fields
ASSIGN ALL	Assign or edit attributes for all fields in the form
ASSIGN FIELD *fldname*	Assign or edit attributes for the field named *fldname*

ASSIGN is used to assign field attributes after creating the form's screen image. Any new fields placed in the form may have their field attributes defined by using either ASSIGN or ASSIGN NEW. If you exit from the field attribute assignment phase and then return to change any previously assigned fields, you must use the ASSIGN ALL or ASSIGN FIELD commands.

Fields that have changed their locations as a result of the OPENLINE or DELLINE operations, or as a result of character insertion or deletion on another part of the line, are recognized as existing fields. Fields whose pictures are modified must be redefined.

If you select ASSIGN NEW or ASSIGN ALL, you may proceed to assign attributes to the next field by pressing the ENTER key. To return to the COMMAND: prompt before you have finished all the fields, type the period (.) on the keypad. (Remember that this results in assignment of default values to all remaining fields in the form. The default value for a field name is 6 blanks.) If you used ASSIGN FIELD, the Form Editor returns to the COMMAND: prompt when you press ENTER for that field.

The Form Editor displays the field attribute questionnaire for each field on the form. The TAB key moves the cursor from one field to the next within the questionnaire. Pressing ENTER after going through the questionnaire for the last field in a form causes the Form Editor to reissue the COMMAND: prompt.

NOTE

The assignment of invalid combinations of attributes to a field results in an error message. To continue, you press the ENTER key to redisplay the questionnaire for that field and correct the attribute that caused the error message.

The following field attributes appear in the Field Attributes Questionnaire:

Name Contains the name by which the field is known and referred to by your program. Unique field names are not required if a form is to be accessed by the FDV3GETAL call. However, if the application is to access one field at a time, unique names should be assigned. (If a form contains more than one field with the same name, the Form Driver can access only the first one.) The default value for a field name is 6 blanks.

Right Just If you type a Y, the field is right-justified. If you type an N, the field is left-justified. A right-justified field may not contain a mixed picture. The default is N.

Clear Char The character that you type in this field is displayed in place of the fill character (either zero or blank for the field). For zero-filled fields, it must be a zero. For blank-filled fields, the clear character may be any character; underline, period, and blank are the most common choices. A blank is the default.

Zero Fill If you type a Y, the field is filled with zeroes before the operator enters any data in the field. If you type an N, blanks are stored in the field. Note that the Clear Character attribute must be set to zero if the field is zero-filled. The default is N. The fill character is also returned to the calling program in any positions the terminal operator does not enter data.

Default Specifies the initial value to be stored in the field when the form is loaded by the Form Driver. If you do not respond, the field contains either blanks or zeroes, depending on your response to the Zero Fill attribute. Your answer to Default should be consistent with the picture-validation type of the field. If a default value is not specified, the internal representation of the field is blank or zero-filled depending on the definition. The fill character is always displayed as the clear character. If a field has no default value it is initially displayed with clear characters. The default value may not be longer than the field.

NOTE

The Form Editor does not validate default data values to be certain that they are legal and conform to the picture-validation type of the field.

Help Specifies a line of information associated with the field that the user can read by pressing the HELP key. The help message appears on the last line of the terminal screen. The default is that no help message is displayed. If this field is left blank, the Help form for the entire form is displayed if there is one. Otherwise, the message "NO HELP AVAILABLE" is displayed.

Auto Tab Determines whether entering the last character in the field causes the cursor to advance automatically to the next field. Typing a Y specifies that Auto Tab is in effect. The default is N.

Resp Reqd At least one character that is not the fill character must be entered in the field. If you type a Y, the operator at the terminal must respond to the field with some kind of input before continuing. If you type an N, the operator does not have to respond to the field. The Form Driver uses this attribute to validate the operator's responses for fields. The default value is N.

Must Fill If data is entered in the field, it must be filled so that it does not contain a single fill character. The field must be either empty or full. The default value is N.

A field defined as must-fill but not response-required must be filled by the operator only if he or she enters data in it. It may be left empty.

Fixed Dec If you type a Y, the field is a fixed decimal field, provided that the picture is all 9s with an embedded decimal point. Signed numeric is not valid. If you type an N or if the numeric picture-validation type is not in effect, the field is not fixed decimal. The default value is N.

Indexed This attribute enables you to define identical fields, one below the other, as indexed fields. An N indicates that the field is not indexed. An H indicates that the field is horizontally indexed and that the cursor should proceed horizontally to the next field on the same line in response to the TAB key (or Auto Tab) if the next field is also horizontally indexed. A V indicates that the field is vertically indexed and that the cursor should proceed vertically to the next field in the same column in response to the TAB key (or Auto Tab). The default value is N. The Indexed attribute is illegal for fields in scrolled areas.

Disp Only If you type a Y, only your application program may place data in the field. If you type an N, both the terminal operator and your program may enter data in the field. The default is N.

Echo Off If you type a Y, data in the field is not displayed on the terminal screen. If you type an N, the characters echo as in normal operation. The default is N.

Supv Only If you type a Y, the field is display-only unless the task has turned off supervisor-only mode (by means of a Form Driver call). If you type an N, the field is not display-only and may be accessed by the terminal operator. The default value is N.

The attribute that defines a field as scrolled does not appear on this questionnaire. You can define all fields on a line as scrolled fields by pressing the GOLD/SCROLL key sequence during the EDIT phase.

2.6.10 Assigning Named Data Attributes

Named data is any data that is to be associated with a form but not displayed with it. Usually, named data contains information that the application uses to control program flow in a form-dependent manner. The information may consist of the names of other forms or program modules. Named data also might contain field specific data. Your program accesses named data by means of calls to the Form Driver.

The Entry Form that collects named data consists of the two horizontally-indexed fields of 16 elements. When the Entry Form appears, it includes all existing named data followed by blank named data fields.

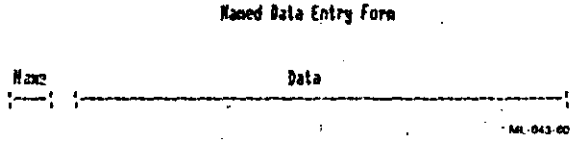
To enter the named data phase, type NAME in response to the COMMAND: prompt. The Form Editor displays the Named Data Entry Form. When the Entry Form appears on the screen, the cursor is at the first character position of the first field. Enter the name by which you want to reference the data that you supply. After you enter the name, press the TAB key to move into the data field. Now enter or edit the named data itself. If a name already exists, tab over to the data field. Exit from the named data phase by pressing the ENTER key.

The Named Data Entry Form (Figure 2-4) has two fields:

NAME A 6-character field that receives the name of the data item. Form Driver calls access an element of named data by using either its name or its index number in the list of named data for a form.

DATA A 60-character field that receives the data.

Figure 2-4: Named Data Entry Form



2.7 A Step-by-Step Example of How to Use the Form Editor

This section presents an example of creating and modifying screen versions of forms. The example demonstrates some of the most common Form Editor commands, functions, and design processes. The example does not cover all Form Editor features, and it is not a complete tutorial. The purposes of this example are as follows:

- To illustrate how you can design a computerized version of a simple printed form.
- To show you what the screen looks like while you are working with the Form Editor.
- To introduce how the Form Editor uses the VT100 special function keypad to control editing functions.
- To introduce how the Form Editor uses the Form Driver and questionnaires to collect information from you about the form that you are designing.

This example has three stages. In the first stage, a printed form is described. Assume that the form was originally designed for a card file of a company's vendors. Before designing the computerized version of the form, read the requirements of the fields in the form section (2.7.1.2).

In the second stage, you create the screen version of the form. Each step in this stage starts with an instruction, and each step completes a part of the exercise of designing the computerized version of the sample form. Read the instruction, and then look at your screen while you follow the instruction. Watch how the Form Editor responds. Finally, read the explanation that follows the instruction.

In the third stage, you modify one of the demonstration forms that you received as part of your FMS software kit. Use the same procedure for the steps in this stage as for the second stage.

You are encouraged to try using this example. You will be able to add the new form that you create and the demonstration form that you modify to the demonstration form library file DEMLIB.FLB. You can then demonstrate how your new form works by running one of the demonstration programs supplied in your VAX-11 FMS software kit.

2.7.1 The Printed Form

The first steps in designing a screen version of the form are:

- Provide an overview or a rough draft of the new form.
- Describe the requirements for each field.
- Describe the layout of the form and any special video features that it is to include.
- Sketch the screen form and include the maximum lengths of fields.

2.7.1.1 Overview of the New Form — The new form will have fields for all of the information that the printed form can contain. This example assumes that the new form will be used only to enter the vendor information that is in a card file.

VENDOR is the name to be assigned to the form. For now the form will not have a help form associated with it. It will use the 80-column screen width and the full screen height (screen lines 1 through 23).

2.7.1.2 Requirements of the Fields in the Original Form — This section describes the requirements for the fields that are in the original form.

1. Vendor Number

   Vendor numbers are in the following form:

   B-67-0095

   The first character can be any letter. Except for two hyphens as shown, the remaining characters must be digits. An operator must enter the vendor number. Programs that use the form can then use the vendor number to get other vendor information from a computer file and display that information.

2. Vendor Name

   Vendor names may be as long as 38 characters and may include any printable character. When the name is first entered, it must be typed exactly as it appears on the file card.

3. Address

   The top line in the address shows the vendor's street address. The next line shows the city and state. The bottom line shows foreign countries and mail codes, such as the ZIP code.

4. Contact

   Contact names are the names of the people in the vendor companies who are most informed about the sample company's business. Contact names may be as long as 28 characters and may include any printable character.

5. Phone

   The form needs to be designed only for one standard North American telephone number in the following form:

   (123) 555-4678

   As shown, parentheses enclose a 3-digit area code. A 7-digit number has a hyphen separating the exchange code from the line number. All input characters must be numbers. The telephone number is not required information, but if a telephone number is entered, all 10 digits must be entered. The area code has the default value 111 because most vendors are in that area, but there is no default value for the balance of the telephone number.

6. Extension

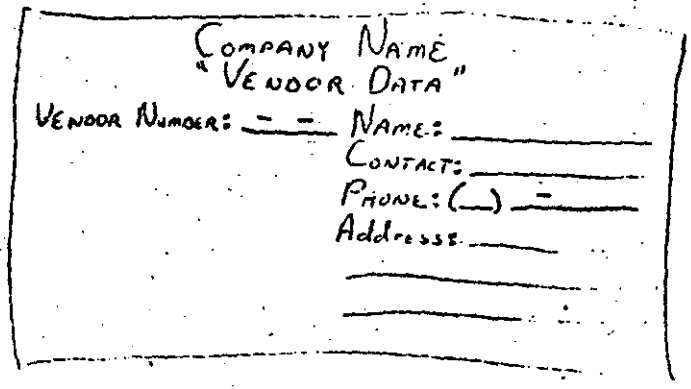
   The form needs to be designed for two telephone extension numbers. To make the new form as flexible as the original printed form, the new form will accept extension numbers up to seven digits, to cover the cases when different vendor extensions are complete 7-digit telephone numbers. The telephone extension is not required information. All input characters must be numbers, but any number of characters is valid.

2.7.1.3 Layout and Video Features of the New Form — The layout of the new form will follow the sketch that appears in Figure 2-5. A screen width of 80 columns will provide ample room. Abbreviations are not necessary. The example assumes that the vendor number is the most important piece of vendor data, and therefore, the sketch shows it at the upper-left corner of the form.

The field will be in reverse video and underlined to show its importance. Other fields will have the bold video attribute to make the values that operators enter more visible. The title of the form, "Vendor Data," will also have the bold video attribute. Field labels will be in standard video. (The bold video attribute really does not look good in a form if used as much as specified here.)

2.7.1.4 Sketch of the Form Named VENDOR — Figure 2-5 is a sketch of the form that you will be creating in this example. Several other designs would be equally effective. In many cases, the sketch that you use may be less detailed than the one in Figure 2-5. Since you can easily change the design by using the Form Editor, you need only enough detail in a sketch to show the number of fields on each line and the rough alignment of fields. More detail appears in Figure 2-5 in order to increase the reliability of this example.

Figure 2-5: Sketch of the Form Named VENDOR



2.7.2 Creating the Screen Form

This section guides you from starting the Form Editor through each of the other steps that you need to complete in order to create a screen version of the form named VENDOR. Each step starts with an instruction. Read the instruction, and then look at your screen while you follow the instruction. Watch how the Form Editor responds. Finally, if an explanation follows the instruction, read it and then go on to the next step. Each explanation begins with the symbol >>>.

The first time you work with this example, follow each instruction carefully. Each step depends closely on the preceding step.

1. Log on to a system that includes the Form Editor. Check with your system manager if you are not sure whether or not the Form Editor is available on your system.

2. Set your VT100 as follows:

   - For the block cursor (▮)
     use SET-UP MODE B.
   - For the 80-column screen width
     use SET-UP MODE A. (This is the default.)
   - For the standard video display (light characters on a dark background)
     use SET-UP MODE B. (This is the default.)
   - For signalling with the terminal bell
     use SET UP MODE B.

   (The *VT100 User Guide* Order #EK-VT100-UG-002 has detailed directions.)

2.7.2.1 Starting the Form Editor—Step 3 —

3. Start the Form Editor by entering the following (The prompts that your system types are in black. The responses that you should type are in red.)

```
$ MCR FED RET

   or

$ FED:==MCR FED RET
$ FED RET
```

In response to the FED> prompt, type:

```
FED> EDIT
```

>>> Each set of commands starts the Form Editor on the corresponding system. The commands also specify that you are developing a screen form. The Form Editor responds by displaying the first questionnaire for a new screen form, the Form Wide Attributes questionnaire.

2.7.2.2 Assigning the Form Name—Steps 4–5 —

4. On the keyboard, type the name VENDOR. If you make a mistake, press the DELETE key to erase incorrect characters and then complete the form name correctly. Later steps depend on the fact that the name of the new form is VENDOR. When the form name is correct, press the RETURN key.

>>> Each character you type appears in the "Form Name" field. The cursor advances through the field from left to right.

When a questionnaire is displayed, press the RETURN key to do the following:

- Assign to the questionnaire whatever value you put into the field.
- Store the questionnaire information internally until you change it or save the form description that you are creating.
- Erase the questionnaire from the screen and respond to a Form Editor command, in some cases.
- Continue a process by changing your display in some way.

With the Form Wide Attributes questionnaire displayed, pressing the RETURN key always causes the Form Editor to erase the screen and display the prompt COMMAND: on the last line.

5. Type the command EDIT. Press the DELETE key to correct mistakes. When you complete the command, use the ENTER function — press the ENTER key on the keypad, or press the RETURN key.

>>> The ENTER or RETURN function key causes the Form Editor to execute the command that you have just typed. When the Form Editor executes the EDIT command, it displays the screen form that you are designing and shows you each detail of the form that you have specified so far. In this case, your new form is entirely blank — 23 lines long, with 80 spaces in each line. The cursor appears in the upper left corner of the screen on Line 1 and Column 1.

While you are editing a form, the Form Editor uses Line 24 to show you information about the cursor's location and several Form Editor settings that you can change while you are editing. At this point, the different sections of Line 24 and their meanings are:

- CURSOR TXT NOR LIN 1 COL 1

  The character that the cursor is on is a text character (TXT) and the line is a normal line of a form (NOR), not a scrolled line. (Scrolling features are explained later in this chapter.) The cursor's position is on Line 1 and Column 1.

- MODES TXT OVS ADV

  The current settings of the editing modes are as follows (later steps

demonstrate the effects of the different modes):

— The text mode (TXT) for entering background text. You cannot use the field mode until you have put in all the background text. The field mode will allow you to assign attributes to each field label you created in text mode.

— The overstrike mode (OVS) for replacing the character that the cursor is on with the character that you type.

— The advance mode (ADV) for advancing the cursor to the right and downward when certain cursor movement functions are used.

NOTE

While using the Form Editor, you will be using the form editor auxiliary keypad as well as the main keyboard to perform specific form editing functions. Refer to the Form Editor keypad layout (Figure 2-1).

2.7.2.3 Creating the Background Text—Steps 6–20 —

6. Use the Downarrow function to move the cursor to Line 2. Press the Downarrow key once.

   >>> The Downarrow function moves the cursor straight down one line at a time. The Form Editor reports the cursor's new position in Line 24.

7. With the cursor on Line 2 and Column 1, type the name of your company or any other company name that you would like to use. Press the DELETE key to correct mistakes.

8. Use the Leftarrow function to move the cursor back to Line 2 and Column 1. Press the Leftarrow key several times.

9. Use the INSERT function to set the Form Editor to the insert mode. Press the 9 key on the keypad.

   >>> The standard function of the 9 key on the keypad is the INSERT function. The function sets the Form Editor to the insert mode. The abbreviation INS replaces OVS in the modes section of Line 24. In the insert mode, the Form Editor moves characters out of the way of insertions rather than replacing the characters.

10. Move the company name to the right in Line 2 by inserting spaces at the beginning of the line. Insert spaces until the company name is centered in Line 2 on column 39 or 40. Hold the space bar down for each space that you want to insert.

11. Use the BLINE function to move the cursor to Line 3 and Column 1. Press the 0 key on the keypad.

    The standard function of the 0 key on the keypad is BLINE. In the advance mode, the BLINE function advances the cursor to the next line and Column 1. In the backup mode, the BLINE function backs up the cursor up to Column 1.

12. Use the following sequence of functions and keyboard keys to move the cursor to Column 34.

    Press the PF1 key on the keypad, then type 33 on the keyboard, and finally press the Rightarrow key:

    GOLD 33 Rightarrow

    >>> The only function on the PF1 key is the GOLD function. When you use the GOLD function before typing a number on the keyboard and then use another Form Editor function, the Form Editor repeats the last function as many times as you have specified. In this case, the Form Editor repeats the Rightarrow function 33 times and the cursor moves from Column 1 to Column 34.

13. Type the title of the form, Vendor Data. Or type any other title that you would like to use. Since the cursor is at Column 34, the title Vendor Data will be centered.

14. Move the cursor back to Line 2. Press the BACKUP key on the editor keypad, then the BLINE key to get to Line 2. Then press the DELCHAR key to remove spaces and to center the company name.

15. Move the cursor to Line 5 and Column 1. To do this, use the ADVANCE BLINE function.

16. With the cursor on Line 5 and Column 1, type the field label for the vendor number field, Vendor Number:.

17. Use the Rightarrow function to move the cursor to Column 34. Press the Rightarrow key and watch the column number in Line 24. You can also press GOLD 19 Rightarrow. Then type the label for the vendor name field, Name:.

18. Use the Downarrow and Leftarrow functions to move the cursor to Line 6 and Column 34, directly under the N of Name in Line 5. Then type the label for the Vendor contact field, Contact:.

19. Use the Downarrow and Leftarrow functions again to move the cursor to Line 7 and Column 34. Then type the label for the vendor telephone field, Phone:

20. Use the Downarrow and Leftarrow functions again to move the cursor down two lines to Line 9 and Column 34. Then type the label for the vendor address fields, Address:

2.7.2.4 Creating the Fields—Steps 21–39 —

21. This and the following steps create the fields whose labels you have typed. Use the BACKUP function to change the directional mode and the BLINE function to move the cursor back to Line 5 and Column 1. Press the 5 key on the keypad. Then the 0 key on the keypad several times until the cursor is back on Line 5.

>>> The standard function of the 5 key on the keypad is the BACKUP function. The BACKUP function sets the Form Editor to the backup mode. The abbreviation BCK replaces ADV in the modes section of Line 24. In the backup mode, the BLINE function backs up the cursor directly to Column 1.

22. Move the cursor to Line 5 and Column 15. Use the OVERSTRIKE function and the FIELD function to set the Form Editor to the overstrike and field modes. Press the following sequences of keys:

    - For the OVERSTRIKE function, the PF1 key and then the 9 key on the keypad.
    - For the FIELD function, the PF1 key and then the 8 key on the keypad.

    >>> The alternate function of each keypad key is the function whose name is at the bottom of the key in the keypad diagram. The alternate function of the 9 key on the keypad is the OVERSTRIKE function, and the alternate function of the 8 key on the keypad is the FIELD function. To use an alternate function, use the GOLD function first and then press the key that controls the function that you want to use.

    The OVERSTRIKE function sets the Form Editor to the overstrike mode, as described earlier.

    The FIELD function sets the Form Editor to the field mode. To create a field, the Form Editor must be in the field mode. In the field mode, you can type only field picture characters and field format characters.

    The full sets of field picture and field format characters are described later in this chapter. In this example, you will need to use only the field characters that are listed in Table 2–4.

Table 2–4: Field Characters Required for the Example

Character	Usage
Field-Picture Characters	
9	For the positions in the vendor number and telephone number where a number is the only valid character.
A	For the first position in the vendor number, where a letter is the only valid character.
X	For the vendor name, contract name, and vendor address fields, where any printable ASCII character is valid.
Field-Maker Character	
(	For enclosing the area code in the telephone number.
)	For enclosing the area code in the telephone number.
-	For separating the two parts of the telephone number.

23. In this example, vendor numbers are in the following form:

    B-67-0085

    To create the field for the vendor number, type A-99-9999.

    >>> A-99-9999 specifies the characters that are valid for each column in the field; they make up a field picture. The picture specifies that the first character in the field must be a letter or a space and the other characters must be digits. The hyphens separate parts of the field. For a program that processes the field, the hyphens will not be part of the field value. Therefore, the program only uses seven characters, although nine are displayed.

24. Move the cursor to Line 5 and Column 43. Create the vendor name field by inserting the letter X 37 times. The easiest way to do this accurately is with the following sequence:

    GOLD 37 X

    Press the PF1 key, type 37 on the keyboard, and press the X key.

    >>> The GOLD function sequence for repeating functions also repeats characters that you want to insert.

    Any character may appear in a vendor name. Therefore, the form has to allow any character.

25. Move the cursor to Line 6 and Column 43. Create the vendor contact field by inserting the letter A 28 times. Use the following sequence:

    GOLD 28 A

    >>> Assume that only spaces and letters can appear in the contact name. Periods (.) after initials and abbreviations will not be copied from the card file. If an operator types a period or other invalid character, the Form Driver will refuse to accept the character and will signal the operator with the following message:

    ALPHABETIC REQUIRED

26. Move the cursor to Line 7 and Column 43 by pressing GOLD 42 Rightarrow. Create the vendor phone field by typing (999)999-9999.

    >>> Assume that only the digits 0-9 can appear in a phone number. When old phone numbers that include letters in the exchange code are copied, the operator will convert the letters to the corresponding numbers.

27. Move the cursor to Line 9 and Column 43 by pressing the Downarrow key twice, and the <Leftarrow> until you see 43 in the Edit Status Display field column on the bottom of the screen. Create the first vendor address field by inserting the letter X 28 times.

    >>> Assume that any character may appear in an address.

28. To experiment with duplicating a field without retyping it, move the cursor back to the first X in the VENDOR address field. Then use the following sequence of functions to erase the field picture and restore it to the form description:

    GOLD DELEOL GOLD UNDELLINE

    Press the PF1 key, the 2 key on the keypad, the PF1 key again, and the PF4 key.

    >>> The alternate functions of the 2 and PF4 keys are DELEOL and UNDELLINE. The DELEOL function erases the cursor's character and the other characters between the cursor and the end of the line. The Form Editor stores the erasure in an internal line buffer, in case you want to restore the last line erasure that you make.

    The UNDELLINE function restores the string that is in the line buffer to the form description. When you want to create several fields with the same field picture, one easy method to use is to create one field picture, erase it, and then restore it in as many positions as needed.

29. Move the cursor to Lines 10, and 11. With the cursor in Column 43 in each line, create one of the vendor address fields by using the UNDELLINE function.

2.7.2.5 Assigning Field Attributes—Steps 30–40 —

30. In this step and the following steps, you will complete the Field Attributes questionnaire for each field that you have created. To begin work with the Field Attributes questionnaire, enter the ASSIGN command. Use the following sequence:

    GOLD COMMAND ASSIGN ENTER (or RETURN)

    >>> The alternate function of the 7 key on the keypad is the COMMAND function. After the COMMAND function, the Form Editor erases Line 24 and displays the prompt COMMAND: . When the prompt appears, enter a command by typing on the keyboard and use the ENTER function to cause the Form Editor to execute the command.

    The ASSIGN command causes the Form Editor to display the Form Attributes questionnaire for each new field. A new field is a field for which no field attributes have been assigned. In this case, all of the fields that you have created are new. The first new field is the Vendor Number field. The Form Editor displays the Form Attributes questionnaire so that you can still see the field itself and then identifies the field by replacing each picture character with an underline character (_). Within the Field Attributes questionnaire, the cursor is displayed in the first field of the questionnaire.

    Like the Form Wide Attributes questionnaire, the Field Attributes questionnaire is also an FMS form that is displayed by the Form Driver. The full set of fields in the Field Attributes questionnaire is explained later in this chapter. For this example, the fields that you need to complete are listed in Table 2-5.

Table 2-5: Field Attributes Required for the Example

Attribute	Usage
Default	To specify the most common area code that occurs in vendor telephone numbers.
Field name	To provide a unique identifier for each field.
Help	To provide reminders to the operator about completing fields.
Must fill	To require the operator to enter all of the characters in the vendor number and telephone number.
Response required	To require the operator to enter the vendor number before finishing with the form.

31. For the Vendor Number field, type the field name NUMBER and press the TAB key to move to the next field in the questionnaire. Press the DELETE key to correct any typing errors.

    >>> When the Form Driver is displaying a questionnaire, the Form Driver displays each character as you type it. The TAB key signals that you are finished with the Name field, although you can return to the field later and change it. The Form Driver responds by moving the cursor to the next field that you should complete. Table 2-6 lists the Form Driver editing functions that you will need in this example. The full set of editing functions is explained in Chapter 4.

Table 2-6: Form Driver Editing Functions Required for the Example

Function	Usage
BACKSPACE	To backup from field to field in a questionnaire.
DELETE	To erase a single character in a questionnaire response.
LINEFEED	To erase an entire questionnaire response.
RETURN	To signal that all responses are correct in a questionnaire.
TAB	To advance from field to field in a questionnaire.

32. Press the TAB key four times. With the cursor at the beginning of the Help field, type a short, helpful message that describes how an operator is to type a vendor number. For example:

    ```
    Copy the vendor number from the old vendor card.
    ```

    Press the DELETE and LINEFEED keys to correct mistakes.

>>> Each time you press the TAB key, the cursor moves to the next field in the questionnaire. For the Right Just, Clear Char, and Zero Fill fields, the default field attributes are unchanged. Therefore, in your new form, the Vendor Number field will have the following corresponding attributes:

- Not right justified.
- The space is the clear character. (It is better to assign a clear character such as underline or if space is used assign the reverse video attribute so the field is visible on the screen.)
- Not filled with zeroes.

The Default field in the questionnaire remains blank. Therefore, in your new form, the Vendor Number field will not have a default value.

33. When you have typed the help message, move the cursor to the Resp Reqd field. Now type Y for "yes".

    >>> In your new form, the Vendor Number is required information. By typing Y, you assigned the response-required field attribute. The Form Driver responds by moving the cursor to the next field — that is, as if you had pressed the TAB key.

34. With the cursor on the Must Fill field, type Y.

    >>> In your new form, the operator response must fill the Vendor Number field. By typing Y, you assigned the Must Fill attribute. The Form Driver responds by automatically moving the cursor to the next field.

35. Press the RETURN key.

    >>> For the field attributes after the Must Fill field, the defaults are correct for the Vendor Number field. The RETURN or ENTER key signals that you are finished with the questionnaire. The Form Driver responds by displaying a fresh image of the Field Attributes questionnaire. The Form Driver also identifies the next field in your new form, the Vendor Name field, as the field to which you should now assign field attributes. The cursor appears at the beginning of the Name field in the questionnaire.

36. Type UNAME as the field name. Move the cursor to the Help field with the TAB key and type a HELP message such as the following:

    ```
    Copy the vendor's name from the old vendor card.
    ```

    >>> The other default attributes are correct for the Vendor Name field. Therefore, press the RETURN key when you complete the HELP message. The Form Driver identifies the next field in your new form, the Contact field, as the field to which you should now assign field attributes.

37. Type CONTAC as the field name. If you want to specify a HELP message for the Contact field, move the cursor to the Help field and type the message. The other default attributes are correct for the Contact field. Therefore, when the Name and Help fields are complete, press the RETURN key.

38. Type the name PHONE for the next field. Move the cursor to the Default field and type 111 as the default area code. Then move the cursor to the Help field if you would like to assign a HELP message for the Phone field. One example of a HELP message is:

    ```
    The area code and a 7-digit number are required.
    ```

    >>> For the new form, the default area code is 111, although the design does not call for a default number. With 111 as the only printing characters in the default value, the field will look like the following example when the Form Driver displays your new form:

    ```
    Phone : (111) ---
    ```

39. Assign the Must Fill attribute to the Phone field. Advance the cursor to the Must Fill field and type Y. The other default field attributes are correct for the Phone field. Press the RETURN key when you have finished assigning the Must Fill attribute.

    >>> The telephone number is not required input data. If, however, the operator types a number, all 10 columns of the area code and number must be complete. The Must Fill field attribute is assigned but the Resp Reqd field attribute is not assigned. Since this field contains data already (the default value), it must be filled unless the default area code is deleted. If a Must Fill field contains any data, it must be filled — as is the case here.

40. For each of the Address fields in your new form, complete the following procedure:

    - Assign field names to each — for example, ADDR1, ADDR2, and ADDR3.
    - Assign a HELP message, if you would like to do so.
    - For the other field attributes the defaults are correct. Press the RETURN key when you finish assigning the field attributes for each field.

    When you press the RETURN key after assigning the field attributes for the last Address field, you have finished assigning attributes to all fields in your new form. The Form Editor will return you to the COMMAND: prompt.

2.7.2.6 Assigning Video Attributes—Steps 41–46 —

41. Assigning video attributes is part of the process of editing a form description. The following steps tell you how to make the following assignments:

    - Display the company name in boldface.
    - Display the Vendor Number field label and field picture in reverse video.

    With the COMMAND: prompt displayed in Line 24, type EDIT.

    >>> The Form Editor responds to the EDIT command by displaying your new screen form.

42. To assign video attributes to character positions in a form, first mark the positions by putting them in a select range. Then assign to the select range the combination of video attributes that you want.

    Move the cursor to the first character of your company name. Now use the SELECT function: press the period (.) on the keypad.

    >>> The Form Editor responds by adding information about your select range to Line 24. When you are building a select range, the Form Editor shows the line and column number of the cursor's position at the time you used the SELECT function.

43. Advance the cursor to the blank that is at the end of your company name. Now use the VIDEO function. Press the 7 key on the keypad.

    >>> The Form Editor displays the `VIDEO:` prompt on Line 24.

44. To assign the bold video attribute to the select range, respond to the `VIDEO:` attribute by typing BOLD and then press the ENTER key.

    >>> The Form Editor displays the select range in boldface and again displays the `VIDEO:` prompt.

45. To finish assigning the video attributes, press the ENTER key without specifying a video attribute. Then advance the cursor to the V of the Vendor Number and begin to build a new select range by using the SELECT function.

    >>> The Form Editor updates the line and column numbers in the select range report in Line 24.

46. Advance the cursor to the blank that follows the field picture (`A-99-9999`) for the Vendor Number field, and use the VIDEO function. When the Form Editor displays the `VIDEO:` prompt, type REVERSE and press the ENTER key. To stop assigning graphic attributes, press the RETURN key again.

    >>> The Form Editor responds by displaying the field label and picture in reverse video.

2.7.2.7 Assigning Named Data—Steps 47–50 —

47. This example assumes that you want to experiment with your new form by having the demonstration program display the form. To make that possible, you must assign named data to your new form. The demonstration program is listed and explained in Appendix A. The named data label that you need to assign is "NXTFRM" and the named data value to be associated with that label is the string ".NONE." Press GOLD COMMAND.

    With the `COMMAND:` prompt displayed on Line 24, type the NAME command and press the ENTER key.

    >>> The Form Editor responds by displaying the Named Data Entry Form. The fields on the left in each line of named data are the fields for a label that are from one to six characters long. On the right is the data string that is from 0 to 66 characters long. The label is simply an identifier by which a program can request (the Form Driver searches — not the program) an associated data string. The cursor is at the beginning of the first field in the questionnaire, the NAME field.

48. To enter the label, type NXTFRM. To move the cursor to the Data field, press the TAB key and then type .NONE.

    >>> As in the other questionnaires, the Form Driver is processing the Entry Form and your responses.

49. Since the form does not require any other named data, press the RETURN key to get the `COMMAND:` prompt.

    >>> When you press the RETURN key while working with the Named Data questionnaire, the Form Editor displays the `COMMAND:` prompt.

50. You have now completed your new computerized version of the sample form. To save the form description that you have created, use the SAVE command. Complete the following sequence:

    ```
    SAVE ENTER
    ```

    >>> In response to the SAVE command the Form Editor saves your new form description in an output file and displays the following message:

    ```
    ?FED-Form being saved
    ```

    The Form Editor's prompt for a command line is then displayed:

    ```
    FED>
    ```

2.7.2.8 Editing One of the Demonstration Forms—Steps 51–57 —

51. With the FED prompt displayed, you can continue to use the Form Editor to work on another form description or you can stop the Form Editor. The following steps assume that you want to have the demonstration program display your new form. For the demonstration program to do that, you must modify the named data for the First form that the demonstration uses. The First form is a menu that is illustrated and explained in Appendix A. The form is named FIRST and is stored in the form library file DEMLIB.FLB. You need to modify the form as follows:

    - Add an alternative exercise to the list in the form by adding the following line of background text:

      ```
      4 Enter vendor data
      ```

    - Add a named data label and value to the other named data that are already associated with the form. The label and value are:

      ```
      5 VENDOR
      ```

To edit the form named FIRST, respond to the FED prompt by typing the following command:

```
FED>DEMLIB.FLB RET
```

When the Form Editor responds with the prompt: `Form name?`, type FIRST and press the RETURN key.

>>> The Form Editor displays the screen image of the form named FIRST and the `COMMAND:` prompt. The form has only one field, the single character field following the word Do. The field picture character 9 specifies that only numeric responses are valid for the field. All other characters in the form are background text.

52. Type EDIT. Then advance the cursor to the E in the line that reads `4 Exit`. With the cursor in that position, replace the word Exit with Enter vendor data. Check the report in Line 24 that you are in the overstrike mode, and type the new phrase.

>>> In the overstrike mode, each character that you type replaces the character at the cursor's position.

53. To restore the choice of exiting from the demonstration program, insert the `Exit` choice. Advance the cursor to the character position directly below the `4` and type `5 Exit`.

54. The demonstration program uses the named data that are associated with the forms in DEMLIB.FLB to transfer control from form to form and to exit. Therefore, you must now change the named data that are associated with the form named FIRST so that:

- The response 5 stops the demonstration.
- The response `4` makes the demonstration display the form named VENDOR and store vendor data in an output file.

To edit the named data associated with the form named FIRST, enter the NAME command. Use the following sequence:

GOLD COMMAND `NAME` ENTER

>>> The Form Driver displays the Named Data Entry Form which has the data and labels that are associated with the form named FIRST. The cursor is at the beginning of the Name field in the first line of the Entry Form.

55. Press the TAB key several times to advance the cursor to the label associated with `.EXIT` (4). Then, press the LINEFEED key to erase the 4, and type 5 to enter the new label.

>>> When the Form Driver is displaying an Entry Form, pressing the LINEFEED key erases the characters in a field. The cursor can be at any character position in the field that you want to erase.

56. For programs that use named data labels to call for data, the named data associated with a form can be in any order. Therefore, to associate the response 4 with the form named VENDOR and the output file for vendor data, you can add the new named data at the end of the original data that is associated with the form. Do the following:

- Press the TAB key until the cursor is in the first blank Name field of the Entry Form.
- Type the label `4`, and press the TAB key to advance the cursor to the Data field.
- Type `VENDOR`, the name of the form that is to be displayed for the response 4 to the form named FIRST. Press the TAB key to advance the cursor to the next blank Name field.
- Type `4F`, a special label that the demonstration program will create, as explained in Appendix A. Press the TAB key to advance the cursor to the Data field.
- Type `VENDOR.DAT` or another file name that you want the demonstration program to use for vendor data.
- To finish editing the named data entry form, press the RETURN key.

>>> When you press the RETURN key while working with the Named Data Entry Form, the Form Editor displays the `COMMAND:` prompt.

57. To save the edited version of the form named FIRST, use the SAVE command. Type `SAVE` and press the ENTER key. The Form Editor saves a form description file named FIRST.FRM and displays this message:

```
?FED-Form being saved
```

2.7.2.9 Storing the New Forms in a Form Library File—Steps 58–59 —

58. The preceding step is the last one in this example that deals with the Form Driver and Form Editor. If you want to experiment with your new form and the edited version of the form named FIRST, you must add the form descriptions to the form library file DEMLIB.FLB. The FMS component that manipulates form descriptions and form library files is the Form Utility (FUT). The Form Utility is described in Chapter 3. The following steps provide the instructions that you need for the forms that you have just edited.

With the Form Editor prompt displayed, stop the Form Editor by typing CTRL/Z. When the system prompt is displayed, start the Form Utility by using one of the following commands or sequences:

```
$ MCR FUT RET
```

or,

create a symbol:

```
$ FUT:==MCR FUT RET
$ FUT
```

59. With the Form Utility prompt (FUT>) displayed, type the following Form Utility command line:

```
DEMLIB.FLB=DEMLIB.FLB/FIRST,FRM:VENDOR.FRM/RPL(9)
```

>>> The Form Utility produces a new version of the form library file DEMLIB.FLB. Because the command line causes the new version of the form named FIRST to replace the original version, the Form Utility reports the full file specification of the replacement. The report is a message that looks like the following:

```
DRA1:[USER]FIRST.FRM;2 Form Name = FIRST
Form replaced
```

The new version of the form library file DEMLIB.FLB contains the edited copy of the form named FIRST and the new form description that you created for the sample vendor data form. You can use this version of DEMLIB.FLB instead of the distributed version when you run the demonstration program.

Chapter 3
The VAX–11 FMS Form Utility (FUT)

The Form Utility is the only program that creates and modifies FMS form libraries. Only the Form Utility should be used when you want to examine FMS library files.

The Form Utility program provides the following services:

1. Extracts and deletes form descriptions from form libraries.
2. Combines form descriptions and form libraries into large form libraries.
3. Converts form descriptions to MACRO-11 object files for applications that use memory-resident forms on PDP-11s. Note: Memory-resident forms are not supported by the VAX-11 FMS Form Driver.
4. Produces printable data descriptions in COBOL format and listing files for form library directories and form descriptions.
5. Creates form libraries from form files.

3.1 Starting and Stopping the Form Utility

3.1.1 Starting the Form Utility

You can run the Form Utility in two ways:

1. `$ MCR FUT` ↵
2. Create a symbol:

```
$ FUT:==MCR FUT ↵
$ FUT ↵
```

The Form Utility starts by displaying the FUT> prompt. Later sections describe how to respond to the prompt.

3.1.2 Stopping the Form Utility

The Form Utility stops in two ways. The way it stops depends on how you start the Form Utility. The two general cases are as follows:

1. When you start the Form Utility with the direct call FUT and include a Form Utility command line. In this case the Form Utility exits after completing the process you have specified and the system displays the system prompt.
2. When you start the Form Utility with the MCR command or with FUT without a command line. In this case the Form Utility remains active after completing a process and displays the prompt FUT>. You can then enter a new file specification string or type CTRL/Z to exit.

3.2 Form Utility Defaults

Table 3-1 summarizes the command default values for the Form Utility.

Table 3-1: Default Values

Item	Default
Input & output UIC	The LOGON directory or the directory specified in the latest SET DEFAULT command.
Input & output volume	The volume installed in the default user device.
Input file name & type	The input file name must be specified. The default input file type is .FLB.
Output file name & type	With the /FF option, no output file name or type can be specified. The form name becomes the file name and the file type is .FRM. With the /FD option, the form name is the default output file name. With the /LI option, the input form library file name is the default output file name. With the /CC option the output extension is .LIB. With other options, the output file name must be specified and the default file types are: .FLB for any output form library file. .FMD for printable form descriptions. .FRM for form files. .LIB for COBOL data descriptions. .LST for form library file directories. .OBJ for PDP-11 object files.
Input file version	The latest version of the input file that is on the input volume.
Output file version	Version 1 for a new file. Otherwise, the Form Utility assigns a version number that is one plus the version number of the latest version that is on the output volume.
Option	The default option is /FD, to produce a printable version of a form description.
Spooling & Block-Alignment	The /-SP and /BA options are the defaults for spooling and block-alignment of form descriptions. The default is to block-align the form library.

3.3 Form Utility Errors

When an error occurs, the Form Utility displays a message and transfers control in one of the three following ways:

1. When recovery is impossible, control transfers to your operating system.
2. When recoverable errors occur in processing form descriptions or files, control transfers to the FUT> prompt.
3. When file specifications and options control transfers to the FUT> prompt.

Appendix B lists Form Utility messages and explains how to look messages up.

3.4 Prompts for Form Library File Processes

The following six options allow you to select individual forms from form library files and process them in different ways:

1. /CC to produce a COBOL data description structure.
2. /DE to delete form descriptions from form library files.
3. /EX to select specific form descriptions from one form library file and store them in a new form library file.
4. /FD, the default option, to produce a printable form description.
5. /FF to select a form description from a form library file and store it in a form description file.
6. /OB to convert form descriptions to PDP-11 object format.

For each of the six options, the Form Utility prompts you for a form name. The general format of the prompt is the full file specification of the form library file followed by the prompt Form name? For example, with DR1:,[USER], and .FLB as the default input volume, DIRECTORY, and form library file type, and with version 6 as the latest version of the form library file DEMLIB, the Form Utility would prompt you as follows:

```
FUT>DESCR.FMD=DEMLIB/FD
DR1:[DIRECTORY]DEMLIB.FLB;6 Form name?
```

You can respond to the Form name?; prompt by typing:

1. A valid form name and pressing the Return key.

   The Form Utility processes only the form description for the form name that you type. It then requests another form name.

2. An asterisk (*) and the Return key.

The Form Utility processes all form descriptions that the form library file contains.

NOTE

Responding with the asterisk is not valid when you have specified the /FF option.

3. The Return key only.

   The Form Utility begins processing the next input file that you have specified, if there is another input file, or stops.

3.5 Form Utility Command Options

This section describes each of the Form Utility command options. The descriptions are arranged in three groups, as follows:

1. Options for control and HELP.

   The /ID option to display the Form Utility identification.
   The /HE option to display the Form Utility HELP file.
   The /SP and /-SP options to control spooling of the files to the line printer.
   The /LI option to list the names of forms in form library files.

2. Options for creating form library files.

   The /BA and /-BA options to control form description block alignment.
   The /CR option to create a form library file by combining files.
   The /DE option to delete form descriptions from files.
   The /EX option to extract form descriptions from files.
   The /RP option to update form descriptions in files.

3. Options for processing and converting form descriptions.

   The /CC option to create COBOL data declarations for form descriptions.
   The /FD option to create a listing of a form description.
   The /FF option to create a form description file from a form in a library file.
   The /OB option to create PDP-11 object modules of form descriptions.

3.5.1 Options for Control and HELP

3.5.1.1 The /ID Option: Displaying the Form Utility Identification — Use the /ID option by itself in the command line to make the Form Utility display its identification. The identification includes the Form Utility's name (FUT), version number, and patch level.

NOTE

FUT is used throughout this chapter as a symbol for the command string: $ MCR FUT.

The following examples illustrate how the Form Utility responds to the /ID option.

```
$ FUT /ID
FUT V01.00

$
$ FUT
FUT>/ID
FUT V01.00
FUT>
```

3.5.1.2 The /HE Option: Using Form Utility HELP File — Use the /HE option by itself in the command line to have the Form Utility display a short summary of the Form Utility command line forms, as well as a list of the command line options and their meanings.

Figure 3–1 shows how to use the /HE option and includes a copy of the Form Utility help display. Later sections in this chapter present the full details about the other options.

Figure 3–1: The /HE Option and the Help Display

```
$ FUT
FUT>/HE
                HELP FOR FRMUTL V01.00

Command lines:

        output-file = input-file,... /input file/option

Options:

/ID         Print identification on terminal
/HE         Print this help text on terminal
/FD         Write form description (default)
/LI         Write library directory listing
/OB         Write object module of forms
/CR         Create library from libraries and forms
/RP         Replace forms in library
/DE         Delete forms from library
/EX         Extract forms to build library
/CC         Create COBOL form description
/FF         Create a form file from a library form
/-BA        Do not block align forms in library
/SP         Spool listing output to line printer

FUT>
```

ML-043-80

3.5.1.3 The /SP and /-SP Options: Requesting Line Printer Listings — Use the /SP option with one of the following options to direct the Form Utility output to the default line printer on your system:

- The /LI option, for form library file directories.
- The /FD option, for printed descriptions of forms.
- The /CC option, for COBOL data descriptions of forms.

When you use the /SP option, the Form Utility creates the output file and spools the file to your line printer after you specify either the form name or indicate with an asterisk you want all the forms listed.

Use the /-SP option with the same options to prohibit line printer listings. The default option is /-SP.

If no output file is specified in the /SP option, then the default output device is the terminal, not the line printer.

3.5.1.4 The /LI Option: Listing Directories of Form Library Files — Use the /LI option to create a printable file that lists the names of the forms that are in form library files. The output file includes the following information:

1. The Form Utility identification and the current date.
2. The full file specification for the form library file.
3. The date the form library file was last updated.
4. The size of the directory within the form library file.

   The Form Utility creates a one-block form library file capable of storing sixty-two forms. If you are writing a large FMS application that requires more forms, you should create more than one form library file to use in your application.
5. For each form description in the form library file:
   - The form name (as assigned by using the Form Editor).
   - The date the form description was last edited with the Form Editor.
   - The size of the impure area that the form requires in an application.

The following example illustrates the /LI option and the format of the output file that the Form Utility produces. Because the command line also includes the /SP option, the Form Utility spools the output file to the line printer after creating the file.

```
FUT V01.00
4-JAN-80

Library DR1:[DIRECTORY]DEMLIB.FLB;1 created: 4-DEC-78
Directory is 1 blocks long


Form     Date         Impure Area (bytes)

FIRST    4-DEC-79        368
CUSTPR   4-DEC-79        328
LAST     4-DEC-79        278
EMPLOY   4-DEC-79        812
PARTS    4-DEC-79        794
CUSTD    4-DEC-79        812
```


3.5.2 Options for Creating Form Library Files

3.5.2.1 The /BA and /-BA Options: Using Block-Aligned Form Descriptions — Use the /BA option with one of the following options to explicitly align each form description from the beginning of a block on the output mass storage volume (/BA is the default option):

- The /CR option.
- The /DE option.
- The /EX option.
- The /RP option.

The input form library files can be aligned or unaligned.

Block aligned form libraries may result in faster access times for an application. Block-aligned form descriptions require larger form library files than non-block-aligned form descriptions. The maximum increase in form library file size is 1 block for each form description.

In practice, block-aligned form library files are usually used unless space is severely limited. For example, if you are packaging a VAX-11 FMS application with its form library files on diskettes or other media with small capacity, you may want to use non-block-aligned form libraries.

Use the /-BA option with the same options to prohibit block-aligned form descriptions.

Section 3.5.2.2 includes an example of the /BA option.

3.5.2.2 The /CR Option: Combining Form Library Files and Description Files — Use the /CR option to combine input form descriptions into one form library file. The option has no effect on any of the input files. If the Form Utility finds a form name more than once in the input files that you specify, the following message is displayed:

```
FUT - Illegal replacement of form; use /RP
```

The following example illustrates the /CR option. Because the /BA option is also used, the form descriptions in the output file will be block aligned.

```
$ FUT DEPT52.FLB/CR/BA=PROJ01.FLB,EC012.FRM;5,PROJ02.FLB
```

When the Form Utility completes the command, the form library file DEPT52.FLB contains the following form descriptions:

- The form descriptions in the latest version of the form library file PROJ01.FLB.
- The single form description in the form description file EC012.FRM;5.

- The form descriptions in the latest version of the form library file PROJ21.FLB.

3.5.2.3 The /DE Option: Deleting Form Descriptions from Form Library Files — The /DE option lets you delete some form descriptions from form library files and combines the remaining form descriptions into a new form library file.

The Form Utility does not change any of the input files. For each input file that is a form library file, the Form Utility displays the full file specification and prompts you for the names of the forms you want to exclude from the output file.

NOTE

The Form Utility accepts form description files as input files with the /DE option. However, none of the form description files will be combined in the output file. In effect, form description files are ignored in this case.

The following example illustrates the /DE option and responses that exclude two forms from each of the input form library files:

```
$ RUN SYS$
FUT>FILMGD/DE=SLIDE.FLB;4,MOVIE.FLB;6↵
DRI:[DIRECTORY]SLIDE.FLB;4 Form name?FIRST↵
DRI:[DIRECTORY]SLIDE.FLB;4 Form name?SECOND↵
DRI:[DIRECTORY]SLIDE.FLB;4 Form name?↵
DRI:[DIRECTORY]MOVIE.FLB;6 Form name?THIRD↵
DRI:[DIRECTORY]MOVIE.FLB;6 Form name?FOURTH↵
DRI:[DIRECTORY]MOVIE.FLB;6 Form name?↵
FUT>
```

When the Form Utility finishes, the form library file FILMGD.FLB;1 contains the following:

- The form descriptions that are in SLIDE.FLB;4 except for the forms named FIRST and SECOND.
- The form descriptions that are in MOVIE.FLB;6 except for the forms named THIRD and FOURTH.

3.5.2.4 The /EX Option: Extracting a Form Library File — Use the /EX option to extract some form descriptions from form library files and combine them in a new form library file. When you include an input file that is a form description file, the Form Utility adds the form description to the output file.

The Form Utility does not change input files. For each input file that is a form library file, the Form Utility displays the full file specification and prompts you for the names of the forms you want to extract.

In the following example the /EX option is used to extract two form descriptions from each input form library file:

```
FUT>PICHLP/EX=SLIDE.FLB;4,003HLP.FRM,MOVIE.FLB;6↵
DRI:[DIRECTORY]SLIDE.FLB;4 Form name?001HLP↵
DRI:[DIRECTORY]SLIDE.FLB;4 Form name?002HLP↵
DRI:[DIRECTORY]SLIDE.FLB;4 Form name?↵
DRI:[DIRECTORY]MOVIE.FLB;6 Form name?003HLP↵
DRI:[DIRECTORY]MOVIE.FLB;6 Form name?004HLP↵
DRI:[DIRECTORY]MOVIE.FLB;6 Form name?↵
FUT>
```

When the Form Utility finishes, the form library file PICHLP.FLB;1 contains the form descriptions for the forms named 001HLP, 002HLP, 003HLP, 004HLP, and 005HLP.

3.5.2.5 The /RP Option: Updating Form Descriptions in Form Library Files — You can use the /RP option to:

1. Replace a form description that is in a form library file with a new version of that form description.
2. Create a form library file that contains all of the forms that are in several form library files.

The Form Utility can process the /RP option if each form that is stored in a form library file has a unique name. The Form Utility processes the input files one at a time from left to right. For input form descriptions with unique form names, the output form library file includes each one. For input form descriptions with the same form names, the output form library file includes only the last one processed. Therefore, the final contents of the output form library file in some cases depend on the order you use when typing the input file specifications.

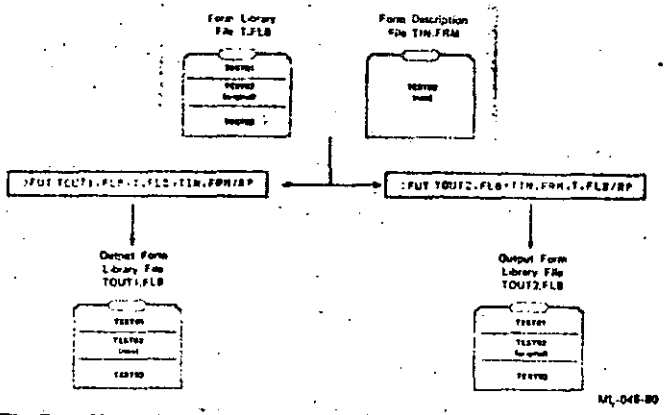
Figure 3-^ illustrates how the final contents of a form library file are different for two Form Utility commands. In the first case, the last version of the form named TEST02 that the Form Utility processes is in the input file TIN.FRM, and the output form library file includes only that version of TEST02. In the second case, the last version of TEST02 that the Form Utility processes is in the input file T.FLB, and the output form library file includes only that version.

3.5.3 Options for Processing and Converting Form Descriptions

3.5.3.1 The /FF Option: Creating Form Description Files from Form Library Files — Use the /FF option to extract a form description from a form library file and store the description in a form description file. With this option, you may not specify an output file name and you may not extract more than one form description at a time. To create a form description file from a form library file, use the general form:

form-library-file-spec/FF↵

Figure 3-2: The /RP Option: Effects of Input File Order on Output File Contents



ML-048-80

The Form Utility displays the full form library file specification and prompts you for the name of the form that you want to extract. The Form Utility then uses the form name that you specify as an output file name, adding the file type .FRM.

In the following example, the /FF option extracts a form description from the input form library file SYS$DISK:FRMLIB.FLB;5 in the directory [DIRECTORY]. The Form Utility creates RY]. The Form Utility creates the form description file named HELP1.FRM.

```
$ FUT
FUT>FRMLIB.FLB;5/FF
DR1:[DIRECTORY]FRMLIB.FLB;5 Form name? HELP1
FUT>
```

3.5.3.2 **The /OB Option: Creating MACRO-11 Object Modules for Forms** — Use the /OB option to convert form descriptions to PDP-11 MACRO-11 object format. You can then build the object files with your FMS application to add memory resident forms to the application.

The Form Utility processes the input files in the order that you type the file specifications. For each input form library file, the Form Utility prompts you for the names of the forms you want to convert. Conversion is automatic for each input form description file.

For each form that you specify, the Form Utility creates an object module with the following two program sections PSECTs:

1. $ FIDX$

   Contains the name of the form and a pointer to the beginning of the form's data structure.

2. $ FORM$

   Contains the form description, including display attributes, default field values, named data, and field help.

The format used by the Form Utility for the object module is the same as for the following MACRO-11 module:

```
        .TITLE   frmnam          !Title of the module
        .IDENT   /V01.00/        !Form Utility version number
        .PSECT   $ FIDX$,D,C0L   !Index that the Form Driver uses
        .RAD50   /frmnam/        !to find the data structure for
                                 !a form that is called by name
        .WORD    FSTR            !Pointer to the beginning of the
                                 !form data structure
        .PSECT   $ FORM$,D       !Form data structure PSECT
FSTR:   .        !Form
          .                      !      data
          .                      !           structure
        .END
```

When you specify more than one form name, the Form Utility converts each form description and produces a concatenated object module of all the forms.

The following example illustrates the /OB option and responses to convert one form description from the form library file DR0:[DIRECTORY]BILL.FLB and the form description file DR0:[30,10]BILHLP.FRM to object format:

```
>FUT BILFRM.OBJ=DR0:[30,10]BILL.FLB,DR0:BILHLP.FRM/OB
DR0:[30,10]BILL.FLB;6 Form name? STARTUP
DR0:[30,10]BILL.FLB;6 Form name?
MCR>
```

The build command file order is important:

```
HLLCBL,FDVLRM/LB:FDVDAT,BILFRM.OBJ,FDVLRM/LB
```

The Form Driver data module (FDVDAT) must come before any memory-resident forms to be included in the program.

3.5.3.3 **The /CC Option: Producing COBOL Data Declarations for Forms** — Use the /CC option to produce an ASCII file that contains the data declaration statements that COBOL applications require for forms. You can then use the COBOL COPY statement in the data division of your COBOL program to refer to the files that contain the data declaration statements. Or you can use a text editor to add the data declaration statement file to your COBOL data division.

For each form that you specify, the Form Utility produces a three-level COBOL structure in the Terminal Format as illustrated in Figure 3-3. COBOL also supports the Conventional (ANS) Format. You can use the COBOL REFORMAT utility to convert the Form Utility's data declaration structure to the Conventional Format.

Figure 3-3 shows a three-field form named PARTS and the COBOL data declaration structure that the Form Utility produces for the form. Assume that the field names PARTNO, DESCRP, and SUPPLR were assigned by using the Form Editor and that the "Suppliers" field has been designed as a vertically indexed field.

Figure 3-3: The /CC Option: Illustration of the COBOL Data Description

```
* COBOL Form Library Structure
*       This structure contains three types of data items:
*       Form Names, prefixed with "FORM-"
*       Names, prefixed with "N-", and
*       Data, prefixed with "D-".
01 FORM-PARTS-DEF.
  03 FORM-PARTS PIC X(6) VALUE "PARTS ".
     03 N-PARTS-PARTNO PIC X(6) VALUE "PARTNO".
     03 D-PARTS-PARTNO PIC X(9).
     03 N-PARTS-DESCRP PIC X(6) VALUE "DESCRP".
     03 D-PARTS-DESCRP PIC X(26).
     03 N-PARTS-SUPPLR PIC X(6) VALUE "SUPPLR".
     03 D-PARTS-SUPPLR PIC X(25) OCCURS 3 TIMES.
```

ML-047-80

The following example illustrates the /CC option and responses to produce concatenated COBOL data descriptions for the form that is in the form description file SYS$DISK:[DIRECTORY]HELP04.FRM;2 and for the forms in the form library file DR1:[DIRECTORY]FRMLIB.FLB;1.

```
$ FUT
D_LIB=[DIRECTORY]HELP04.FRM;2,[DIRECTORY]FRMLIB.FLB;1/CC
DR1:[DIRECTORY]FRMLIB.FLB;1 Form name? *
FUT>
```

NOTE

1. If the same name is used for more than one field in a form, the COBOL compiler will flag one of the fields as an error.
2. A COBOL data declaration cannot be created for a form description that contains blank field names.

3.5.3.4 The /FD Option: Producing Form Descriptions for Printed Listings — Use the /FD option to produce an ASCII file that describes all of the features of a form. You can print the file that the Form Utility produces or display it on your video terminal.

The description produced by the Form Utility is arranged in five major sections:

1. The form description header

   This section lists all form-wide information. For example, the section lists the form name, the associated HELP form name, and the impure area size that the form requires.

In unusual cases, the Form Utility may also detect a problem with a form that the Form Editor did not detect. In such a case, the Form Utility prints a brief summary of each possible problem in this section.

2. The image map

   This section shows all of the constant text in the form and the default value that has been assigned to each field. When no default has been assigned, the image map shows the clear character that has been assigned.

3. The video attributes map

   This section shows the video attributes of all constant text and fields in the form.

4. The field descriptions

   For each field in the form, this section lists the field name, length, position, picture, clear character, and other assigned features.

5. The named data map

   This section includes a full list of the names, associated data, and order of the named data that have been assigned to the form.

The following example illustrates the /FD option and responses to produce descriptions for one form from each of two form library files:

```
$ FUT
FUT>FD.INT=[DIRECTORY]AM.FLB
DR1:[DIRECTORY]AM.FLB;1 Form name? CHILA
DR1:[DIRECTORY]AM.FLB;1 Form name? 
DR1:[DIRECTORY]AM.HLP;1 Form name? CHI01 Form name? CHI01
DR1:[DIRECTORY]AM.HLP;1 Form name? 
FUT>
```

The following sections describe each section of the output file that the Form Utility creates when you use the /FD option.

The /FD Option: The Form Description Header

Figure 3-4 shows an example of the form description header in a form description.

Figure 3-4: The /FD Option: The Form Description Header

```
Form name:              BYE
Help form name:         BYEHLP
First line:             1
Last line:              23
Date created:           28-DEC-78
Owner ID:               0
Form length:            1878 bytes
Number of fields:       4
Impure area size:       2010 bytes
```

ML-048-80

The individual lines in the form description header provide the following information:

- Form name

  As assigned by completing the form-wide attributes questionnaire in the Form Editor.

- Help form name

  As assigned by completing the form-wide attributes questionnaire in the Form Editor.

- First line

  As assigned by completing the form-wide attributes questionnaire in the Form Editor.

- Last line

  As assigned by completing the form-wide attributes questionnaire in the Form Editor.

- Date created

  The most recent date on which the form was processed with the Form Editor.

- Owner ID

  Reserved for future use.

- Form attributes

  If the reverse screen, current screen, and wide screen attributes have been selected in the form-wide attributes questionnaire, they are reported on this line.

- Form length

  As reported in the form-wide attributes questionnaire in the Form Editor.

- Number of fields

  The number of fields with different names. Each occurrence of a scrolled or indexed field is counted.

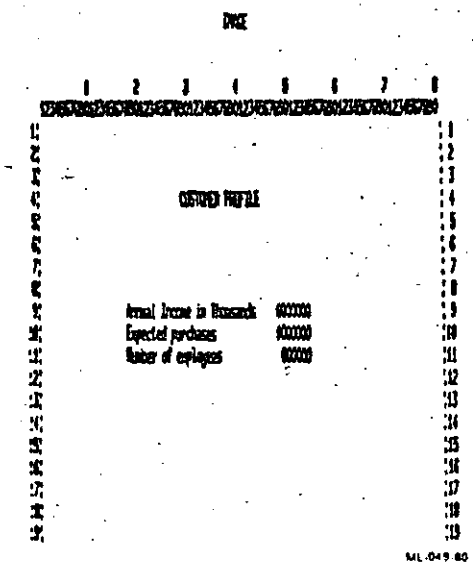
- Impure area size

  As reported in the form-wide attributes questionnaire in the Form Editor.

The /FD Option: The Image Map

Figure 3-5 shows an example of the 80-column image map that the Form Utility produces in a printable form description. (Although the Form Utility shows all 24 lines in the image map, the figure has been compressed for printing in this manual.)

Figure 3-5: The /FD Option: The Image Map



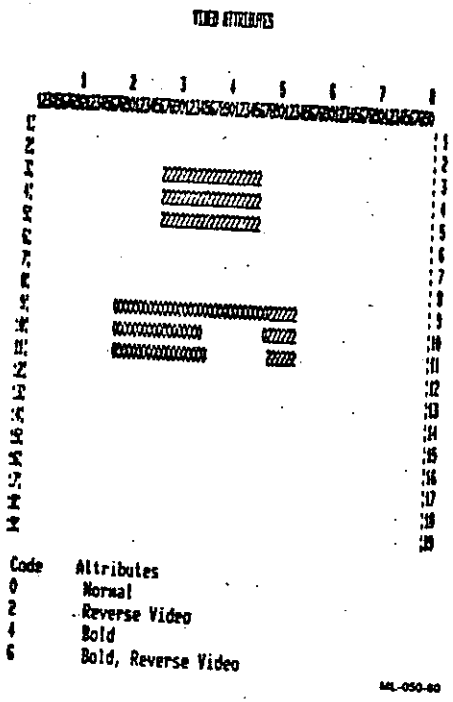
For 80-column forms, the borders of the image map include scales that show the line and column numbers for the map. For 132-column forms, the line numbers do not appear. Except for the video attributes of the form, the image map shows the form as the operator will see it before the Form Driver or the operator enter information in any fields. Each character of the constant text appears in the correct line and column position. Each field appears in the image map with the clear character that was assigned by using the Form Editor, and each field includes any field marker characters such as the hyphen (-).

The /FD Option: The Video Attributes Map

Figure 3-6 shows an example of the 80-column video attributes map that the Form Utility produces in a printable form description. (Although the Form Utility shows all 24 lines in the map, the figure has been compressed for printing in this manual.)

For 80-column forms, the borders of the video attributes map include scales that show the line and column numbers for the map. For 132-column forms, the line numbers do not appear. Within the map, a one-digit or one-letter code for the video attributes of each character appears at the character's position. Table 3-2 contains a complete list of the codes and their meanings. In each map, the codes that actually appear are described below the map under the heading "Key to Video Attributes."

Figure 3-6: The /FD Option: The Video Attributes Map



The /FD Option: Field Descriptions

Figure 3–7 shows an example of the field description that the Form Utility produces in the printable form description.

Figure 3-7: The /FD Option: Field Descriptions

```
                    Field Descriptions

 9 43   Field INCOME of length 6
        Display attributes: Right Justified, Zero Fill
        Field Type: Numeric
        Clear character: '0'
        Picture value: '999999'

                                                    ML-051-80
```


Table 3-2: The /FD Option: Video Attributes Codes and Meanings

Code	Meaning
0	Normal
1	Underline
2	Reverse video
3	Underline, Reverse video
4	Bold
5	Bold, Underline
6	Bold, Reverse video
7	Bold, Reverse video, Underline
8	Blinking
9	Blinking, Underline
A	Blinking, Reverse video
B	Blinking, Underline, Reverse video
C	Blinking, Bold
D	Blinking, Bold, Underline
E	Blinking, Bold, Reverse video
F	Blinking, Bold, Reverse video, Underline

The individual lines in each field description provide the following information:

- Field name, size, and position

  The first line of the field description describes the starting position of the field in terms of the row and column numbers for the first character ("1 39" in the example above). The line also provides the field name and the length of the field as used by the application.

- Display attributes

  Any of the following field attributes, as assigned with the Form Editor:

  — Autotab
  — Display Only
  — Fixed Decimal
  — Full Required
  — Vertical (indexed)
  — Horizontal (indexed)
  — No Echo
  — Right Justified
  — Some Required
  — Supervise Protect
  — Zero Fill

- Field Type

  The type of characters that an operator can enter in the field, corresponding as follows with the field picture characters that the Form Editor accepts:

  — 9 Numeric type
  — A Alphabetic type
  — C Alphanumeric type
  — N Signed numeric type
  — X Any printing character

  The other field type features listed in this section are:

  — Indexed
  — Mixed picture
  — Scrolled

- Clear character

  As assigned with the Form Editor field attributes questionnaire.

- Help text

  As assigned with the Form Editor attributes questionnaire.

- Picture value

  As entered in the Form Editor's field mode.

The /FD Option: The Named Data Map

Figure 3-8 shows an example of the named data map that the Form Utility produces in the printable form description.

Figure 3-8: The /FD Option: The Named Data Map

```
                    Named Data Information

Name          Data

ONE           210 295  23   6 171   83 END
ONEOUT        SYS$DISK:[DIRECTORY]STWAYS.DAT
```

ML-032-88

Chapter 4
Introduction to the VAX–11 FMS Form Driver (FDV)

The Form Driver is a library of routines that is a subcomponent of your program. In an application that uses video images of forms on the terminal screen, using the Form Driver can reduce your programming effort by manipulating the screen, checking responses that an operator types, and displaying help messages and forms when the operator requests them.

This chapter discusses how the Form Driver interacts with:

- The form description, which is created with the Form Editor.
- The terminal operator, who completes the fields in a displayed form.

Throughout the chapter, programming requirements are mentioned and specific subroutine calls are mentioned occasionally but not fully described. Chapter 5 details the programming requirements for different high-level languages. In Chapter 6, the calls are arranged in alphabetical order and a full description is presented for each one. In Chapter 7, programming techniques are described for some typical Form Driver applications.

4.1 Form Driver Interaction with the Form Description

This section describes how the Form Driver uses forms to display information for the operator, guide the operator through a form, and collect the responses that the operator types. The term "form" refers to the image that the operator sees and to the computerized form description that the Form Driver handles internally.

4.1.1 Program Access to Forms

Your program uses form descriptions by reading them from a form library file that has been stored on a mass storage volume, such as a disk.

Form descriptions created with the Form Editor are processed by the Form Utility. For example, after using the Form Editor to create a form description, you must use the Form Utility to store the description in a form library file. Chapter 2 describes how to use the Form Editor, and Chapter 3 describes how to use the Form Utility. Chapter 8 describes the build procedure for each language.

```
DECLARE FDV$RETN  ENTRY (CHAR(*),CHAR(*),FIXED BIN(31))
                  OPTIONS(VARIABLE) RETURNS(FIXED BIN(31));
DECLARE FDV$SHOW  ENTRY (CHAR(*),FIXED BIN(31))
                  OPTIONS(VARIABLE) RETURNS(FIXED BIN(31));
DECLARE FDV$SOFF ENTRY
                  RETURNS(FIXED BIN(31));
DECLARE FDV$SPON  ENTRY
                  RETURNS(FIXED BIN(31));
DECLARE FDV$STAT  ENTRY (FIXED BIN(31),FIXED BIN(31))
                  OPTIONS(VARIABLE); /*The only non-function*/
DECLARE FDV$TERM  ENTRY (FIXED BIN(31))
                  RETURNS(FIXED BIN(31));

/*    These are definitions for form driver completion returns when the */
/*    form driver routines are called as functions. The returns for     */
/*    FDV$STAT are different and are given as comments.                 */

DECLARE FDV$_ITT GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -25 Invalid terminal type */
DECLARE FDV$_FVM GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -23 Error freeing virtual memory */
DECLARE FDV$_ICH GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -6  Invalid channel number specified */
DECLARE FDV$_IFN GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -19 Invalid call in current form context */
DECLARE FDV$_IMP GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -2  Impure area too small */
DECLARE FDV$_INC GLOBALREF VALUE FIXED BINARY STATIC;
                  /* 2   Current form incomplete */
DECLARE FDV$_INI GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -21 Impure area not initialized for call */
DECLARE FDV$_IOL GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -4  Error opening form library */
DECLARE FDV$_IOR GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -18 Error reading form library */
DECLARE FDV$_IVM GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -24 Insufficient virtual memory */
DECLARE FDV$_LIN GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -10 Invalid first line number to display form */

DECLARE FDV$_NOF GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -12 No fields defined for form */
DECLARE FDV$_NSC GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -14 Specified field not in scrolled area */
DECLARE FDV$_STR GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -22 Invalid string length */
DECLARE FDV$_SUC GLOBALREF VALUE FIXED BINARY STATIC;
                  /* 1   Successful completion */
DECLARE FDV$_UTR GLOBALREF VALUE FIXED BINARY STATIC;
                  /* Undefined field terminator */
DECLARE FDV$_ARG GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -20 Invalid number of arguments in call */
DECLARE FDV$_OLN GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -16 Output data too long; truncated to fit */
DECLARE FDV$_ONM GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -15 Specified name data does not exist */
DECLARE FDV$_OBP GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -13 Invalid request to read output only field */
DECLARE FDV$_FCO GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -1  Invalid function code */

DECLARE FDV$_FCH GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -7  Form library not open on specified channel */
DECLARE FDV$_FLB GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -5  Specified file is not a form library */
```



```
DECLARE FDV$_FLD GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -11 Specified field does not exist */
DECLARE FDV$_FNM GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -9  Specified form does not exist */
DECLARE FDV$_FRM GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -8  Invalid form definition */
DECLARE FDV$_FSP GLOBALREF VALUE FIXED BINARY STATIC;
                  /* -3  Invalid file specification */

DECLARE FDV$K_FT_ATB   FIXED BINARY STATIC INITIAL (3);
                    /* Auto tab field was filled */
DECLARE FDV$K_FT_KPD   FIXED BINARY STATIC INITIAL ('0000002C'BX);
DECLARE FDV$K_FT_NTR   FIXED BINARY STATIC INITIAL (0);
                    /* Enter key input */
DECLARE FDV$K_FT_NXT   FIXED BINARY STATIC INITIAL (1);
                    /* Next field key terminated input */

DECLARE FDV$K_FT_PRV   FIXED BINARY STATIC INITIAL (2);
                    /* Previous field key terminated input */
DECLARE FDV$K_FT_SBK   FIXED BINARY STATIC INITIAL (9);
                    /* Scrolled area input terminated with
                       scroll backward key */
DECLARE FDV$K_FT_SFW   FIXED BINARY STATIC INITIAL (8);
                    /* Scrolled area input terminated with
                       scroll forward key */
DECLARE FDV$K_FT_SNX   FIXED BINARY STATIC INITIAL (6);
                    /* Last field in scroll line was terminated
                       with next field key */
DECLARE FDV$K_FT_SPR   FIXED BINARY STATIC INITIAL (7);
                    /* First field in scroll line was terminated
                       with previous field key */
DECLARE FDV$K_FT_XBK   FIXED BINARY STATIC INITIAL (4);
                    /* Input in scrolled area terminated with
                       exit scrolled area backwards key */
DECLARE FDV$K_FT_XFW   FIXED BINARY STATIC INITIAL (5);
                    /* Input in scrolled area terminated with
                       exit scrolled area forwards key */

/*    End of FDVDEF.PLI                                                 */
```

5.2 The Role of the Field Terminators

The field terminators define one of the following conditions:

1. When the operator wants to work on the next form.
2. When the operator wants to work on a different field from the current field.

Each of the keys listed in Table 5–2 controls a field terminator. The Autotab field attribute also controls a unique terminator. When an operator presses a key or completes a field that has the Autotab attribute, the Form Driver either processes the terminator itself and displays the effect for the operator or returns a unique field terminator code to your program and leaves the choice of processes to the program. Table 5–2 also lists the process and code that the Form Driver uses for each field terminator key.

When you set the VT100 keypad to the alternate keypad mode, the Form Driver also treats the keypad's numeric keys, comma (,) key, hyphen (-)

... and decimal point (.) key as field terminators. The codes for these alternate keypad mode terminators are always returned to your program immediately.

Table 5-2: Field Terminator Keys, Codes, Symbols, and Typical Effects

Key	Code (Decimal)	Symbol	Usage or Meaning
ENTER or RETURN	0	FDV$K_FT_NTR	Terminates all entries in the form. If the call being processed is an FDV$GETAL and required entries are not complete, the Form Driver refuses to accept the terminator, and the operator remains in control. If required entries are complete, the terminator is always returned to the program. Therefore, the final effect depends on the next call that the program initiates for an operator response. If any other call is being processed, only the requirements for the current field must be satisfied. If so, control is returned to the program.
TAB	1	FDV$K_FT_NXT	Valid only when the current field is not the last field in the form that is not Display Only. Moves the cursor to the initial position of the next field. Processed by the Form Driver for the FDV$GETAL and FDV$INLN calls and, until an entry is typed or modified, for the FDV$GETAP call. Returned to the program for the FDV$GET call and, after an entry is typed or modified, the FDV$GETAF call.
	6	FDV$K_FT_SNX	Scroll forward to the next field. The TAB key terminated input in last field of a scrolled line. Always returned to the program.
BACKSPACE	2	FDV$K_FT_PRV	Valid only when the current field is not the first field in the form that is not display only. Moves the cursor to the initial position of the previous field. Processed as for the TAB key.

(continued on next page)

Table 5-2 (Cont.): Field Terminator Keys, Codes, Symbols, and Typical Effects

Key	Code (Decimal)	Symbol	Usage or Meaning
	7	FDV$K_FT_SPR	Scroll backward to the previous field. The BACKSPACE key terminated input in the first field in a scrolled line. Always returned to the program.
None (Autotab)	3	FDV$K_FT_ATB	Processed as for the TAB key.
PF3 (Exit Scrolled Area Backward)	4	FDV$K_FT_XBK	(Valid input only when the current field is in a scrolled area) Moves the cursor out of the scrolled area to the initial position of the previous field that the operator is allowed to complete.
PF3 (Exit Scrolled Area Forward)	5	FDV$K_FT_XFW	(Valid input only when the current field is in a scrolled area) Moves the cursor out of the scrolled area to the initial position of the next field that the operator is allowed to complete.
Downarrow (Scroll Forward)	8	FDV$K_FT_SFW	(Valid input only when the current field is in a scrolled area). The scrolled area is scrolled up and the current line remains the same physical line (with new data) or the cursor moves down one line and that line becomes the new current line. The cursor moves to the initial position of the first field that the operator is allowed to complete in the current line.
Uparrow (Scroll Backward)	9	FDV$K_FT_SBK	(Valid input only when the current field is in a scrolled area). The scrolled area is scrolled down and the current line remains the same physical line (with new data) or the cursor moves up one line and that line becomes the new current line. The cursor moves to the initial position of the first field that the operator is allowed to complete in the current line.

This section describes how your program can use the field terminators and Form Driver calls to guide an operator through the fields in a form in any order.

1. Using the FDV$GETAL call.
   a. The program initiates the FDV$GETAL call.
   b. The operator uses the field terminator keys that move the cursor from field to field at any time. The Form Driver processes these field terminators without returning them to the program.
   c. When the operator presses the ENTER or RETURN key, the Form Driver returns the field terminator code and the string of field values to the program.
   d. The program then is in control of what the operator does next.
2. Using a series of FDV$GET calls
   a. The program initiates the FDV$GET call. The operator can only type and change the entry in the specified field.
   b. When the operator presses a field terminator key, the Form Driver returns the field terminator code and the single field value to the program. The program then is in control of what the operator does next. For example, on the basis of the field value or the field terminator, the program can specify the same field or another field in the next FDV$GET call.

5.2.1 Relationship Between the Field Terminators and Form Driver Calls

In effect, the Form Driver works between the operator and the application program that the operator is using. When the program initiates a call to get an operator response, the Form Driver allows the operator to type an entry in a field. When the operator presses a field terminator key that completes the call, the Form Driver passes the response and the field terminator code to the program and prohibits the operator from further typing.

Only the following four Form Driver calls allow the operator to respond:

- FDV$GET, to get the value for a specified field and the field terminator.
- FDV$GETAF, to get the value for any field that the operator chooses, as well as the field name and the field terminator.
- FDV$GETAL, to get a concatenated string of all field values for the current form and the last field terminator used.
- FDV$INLN, to get a concatenated string of the field values from the current line of the specified scrolled area and the last field terminator used.

For each of these four calls, the Form Driver validates all field terminators. For example, with the cursor in the first field in a form, the Form Driver accepts the field terminator for the TAB key but does not accept the field terminator for the BACKSPACE key.

Table 5–3 lists the four calls and shows the field terminator keys that completes each call. The FDV$GET call leaves total control of responding to any field terminator to the program. The FDV$GETAF call allows the operator to choose one field but returns control to the program as soon as the operator completes an Autotab field or modifies a field and presses any field terminator key. The FDV$GETAL call leaves the Form Driver in control of responding to any field terminator except when the operator presses the ENTER or RETURN key. The FDV$INLN call leaves the Form Driver in control within a line of a scrolled area.

For a general illustration of the flexibility that the set of field terminator features and related calls gives you, compare the following two methods of getting all of the current field values from the operator. (The illustration assumes that none of the fields has any special attributes, such as the Response Required attribute.)

Table 5–3: The Relationship Between the Calls to Get Operator Responses and the Field Terminators

Call	Field Terminator Keys That Complete the Call
FDV$GET	Any valid field terminator key or the Autotab code.
FDV$GETAF	ENTER, RETURN, or any typed field entry followed by any valid field terminator key or the Autotab code.
FDV$GETAL	ENTER or RETURN.
FDV$INLN	Any valid field terminator key or the Autotab code.

In terms of designing forms and programs for FMS applications, the following principles provide a useful summary of Tables 5–2 and 5–3:

1. Except for the ENTER, RETURN, PF3, and PF4 keys, the effects of the field terminator keys cannot be changed from what DIGITAL has designed in the following cases:
   a. For the FDV$GETAL call.
   b. For the FDV$INLN call.
   c. For the FDV$GETAF call before the operator makes a field entry.
2. When the operator presses the ENTER, RETURN, PF3, or PF4 key, or, in response to the FDV$GET call, any field terminator key, the program alone controls the effect that the operator sees.

For example, if you use the FDV$GETAL call in a program, the TAB key will always advance the cursor from field to field according to the default order that DIGITAL has implemented. However, if you use a series of FDV$GET calls instead of the FDV$GETAL call, the program is passed the field terminator code for the TAB key and can react to it in any way you specify.

You can, for example, use the FDV$PFT call. After the operator uses any field terminator that returns control to the application program, the program can

initiate the FDV$PFT call, in effect making the Form Driver display the effects of any field terminator key. In the example of an FDV$GET call terminated by pressing the TAB key, the program can react by specifying the BACKSPACE key code in the FDV$PFT call. Then, the effect of the next FDV$GET call would be to move the cursor back to the previous field in the form.

Or you can use another FDV$GET call. Again in the example of an FDV$GET call terminated by pressing the TAB key, the program can react with another FDV$GET call that specifies by name the next field that the operator is to complete, regardless of where the field appears on the operator's screen.

5.2.2 Using the Alternate Keypad Mode Terminators

Normally, the numeric and punctuation keys on the VT100 keypad produce the same numbers and characters that the corresponding keyboard keys produce. Therefore, for many common applications the operator can enter numeric data by using the keypad rather than the more cumbersome keyboard arrangement.

For special applications, you can set the VT100 to the alternate keypad mode from your program and then design the applications to use the numeric and punctuation keys on the keypad as field terminator keys. In this case, the Form Driver always passes the alternate keypad mode terminators to the program immediately, regardless of whether the Input Required and Must Fill requirements are satisfied for the form. The *VT100 User Guide* describes how to set the alternate keypad mode. Table 5–4 lists the keypad keys that are affected by the alternate keypad setting and the code that is returned to your program for each key.

In each case, the character returned is the last character in the escape sequence generated by the key in alternate keypad mode.

Table 5–4: Alternate Keypad Mode Field Terminator Keys and Codes

Keypad Key	Code Returned Character	Code Returned Value (Decimal)
Comma (,)	l (lowercase L)	108.
Hyphen (-)	m	109.
Decimal (.)	n	110.
0	p	112.
1	q	113.
2	r	114.
3	s	115.
4	t	116.

(continued on next page)

Table 5–4 (Cont.): Alternate Keypad Mode Field Terminator Keys and Codes

Keypad Key	Code Returned Character	Code Returned Value (Decimal)
5	u	117.
6	v	118.
7	w	119.
8	x	120.
9	y	121.

5.3 The Impure Area

The size of the impure area must satisfy the requirements of the largest form that you are using if you want your application to migrate to a PDP-11. On VAX/VMS, the Form Driver increases the number of bytes needed dynamically. However, the user should use a size argument that is 1000 bytes. You do this in the call FDV$INIT to the Form Driver. See Chapter 6, Section 6.7.

The impure area is used by the Form Driver to maintain terminal context between calls. If you issue direct calls from your program to display data on or solicit input from the terminal, rather than using the Form Driver for all terminal I/O, the results of the next Form Driver call may not be as expected.

The FDV$INIT call within your program defines the size of the impure area. The actual impure area is dynamically created by the Form Driver. You will notice when running the Form Editor that the fields for impure area and form size are display-only, containing question marks. After you have created a form, you will notice that when you display the Form Wide Attributes questionnaire by typing FORM in response to the command: prompt in FED, that the '???' have disappeared and definitive numbers have appeared there. The impure size specification is available for the programmer to write a transportable program between the RT11, RSX-11M, or VAX/VMS operating systems.

5.4 The High Level Language Interface

The high level language interface, a special component of the Form Driver, processes your high level language Form Driver calls. The interface passes the values that you supply to the Form Driver and returns values to your program from the Form Driver.

The high level language interface is transparent to you and to your program except when you build your FMS application. To use forms, you need to use only the Form Driver calls.

Most of the mutual requirements for the Form Driver and each high level language are the same. They are grouped in the following four categories and described in the sections that follow:

1. The input and output arguments for the Form Driver calls.

2. The syntax of the calls and conventions used in this manual to define the syntax for the different languages.
3. The completion status of calls for success and failure.
4. Interpretation of the field terminators that an operator uses while working with your application and using the terminators flexibly.

5.4.1 General Description of the Arguments

Collectively, the calls use arguments to pass values to the Form Driver and to receive values that the Form Driver returns. For each call, this manual uses the term *Input Arguments* (or *Inputs*) to refer to the arguments that pass values from your program to the Form Driver. The term *Output Arguments* (or *Outputs*) refers to the arguments for values that the Form Driver returns to your program. For example, the FDV$GET call allows an operator to enter data in a field and then returns the field value to the program when the operator finishes. The input arguments for the FDV$GET call are the field name and, if the field is indexed, the field index. The output arguments for the FDV$GET call are the field value when the operator terminated the field and the code for the field terminator.

Table 5-5 shows the abbreviations used for the Form Driver call arguments and describes briefly the requirements or value for each input argument and output argument.

The full descriptions of the Form Driver calls in Chapter 6 also use the abbreviations that appear in Table 5-5.

Table 5-5: Summary of Form Driver Call Argument's Inputs and Outputs

Argument Abbreviation	Requirement or Value
Inputs	
CHAN	An FMS channel number for a form library file, which is mapped to a VMS logical unit number by the Form Driver. It can be any positive integer.
FID	A field name or a named data label, at least 6 characters long. To specify a scrolled area, use the name of any field in the scrolled area.
FIDX	A field index for the specified field (when the field is indexed) or the index for a named data value. The argument is ignored unless the Form Driver is processing an indexed field or accessing named data by index.
FLNM	A form library file specification.
FNAME	A form name, at least 6 characters long.

(continued on next page)

Table 5-5 (Cont.): Summary of Form Driver Call Argument's Inputs and Outputs

Argument Abbreviation	Requirement or Value
FVAL	As an input value, the single value or the concatenated values to be displayed: • in a field. • in the top, bottom, or current line of a scrolled area. • in the last line of the screen. • in an entire form.
IMPURE	The name of a subscripted variable (or array) for the impure area. The VMS Form Driver requires only 8 longwords.
LINE	The explicit starting line number for the form, overriding the line number assigned with the Form Editor.
SIZE	The size of the impure area in bytes. The VMS Form Driver increases the number of bytes needed dynamically; however, to create an area large enough for your program's largest form, make the size 1000 bytes.
TERM	As an input value, the numeric code for the terminator that the Form Driver is to process.
Outputs	
	(The status code is set for all calls.)
FID	The current field name or a named data label.
FIDX	A field index.
FLEN	The length of a specified field (not the length of the data the field contains).
FVAL	A named data value, a single field value, or a concatenated string that is composed of several field values (including padding when a value is shorter than its field).
TERM	The numeric code for the key that the operator used to terminate input: • in a field. • in a line in a scrolled area. • in an entire form.
STATUS	A numeric code for the completion status of the last call that was executed.
STAT2	A numeric RMS status code for detailed information when the STATUS value is -6 (FDV$_IOL) or -18 (FDV$_IOR).

As shown in Table 5-5, the maximum length of form names and field names is six characters. Form and field names that are longer than six characters are truncated when passed to the Form Driver. The field names returned by the Form Driver are six characters long, including any spaces that have been added.

5.4.1.1 Argument Data Types — The data types of the Form Driver arguments depend on the language that you are using. Specific requirements are listed later in this chapter in the sections that provide information on the languages.

The general data types that the Form Driver uses regularly are integers and alphanumeric strings. Examples of arguments that pass integer values to and from the Form Driver are the arguments for:

- The starting line number for a form.
- The code for the key that an operator uses to finish a field.
- The size of the impure area.

Examples of arguments that pass alphanumeric strings to and from the Form Driver are the arguments for:

- The name of a field.
- A named data value.
- A value to be displayed in a field.

5.4.1.2 The Relationship Between Field Lengths and Values — Regardless of the practical purposes of the fields in a form, the Form Driver always treats field values as strings. For example, when the Form Driver returns a field value to your program as an argument to the FDV$GET call, your program receives a string of characters that is as long as the field that you specified. If the value is shorter than the field, Fill Characters (either zeroes or spaces, as you assigned with the Form Editor) are added.

As another example of the relationship between field lengths and values, when you use the FDV$PUTAL call to display specified values in the first three fields of a form, you pass to the Form Driver a concatenated string of the three field values. For any value that is shorter than the field in which it is to be displayed, you add Fill Characters so that the value and the field are the same length.

5.4.2 General Description of Call Syntax

The syntax of the Form Driver calls follows the requirements and conventions of the language that you use. VAX-11 BASIC, COBOL, FORTRAN, and PL/I programs all use the CALL statement. In this manual, only the call statement forms are listed in the detailed descriptions of the calls. For example, the CALL statement syntax for the calls to get a field value from the operator and then display a message on the last line of the operator's screen is as follows:

1. For BASIC and FORTRAN —

```
CALL FDV$GET(fval,term,fidl,fidx])
CALL FDV$PUTL(fval)
```

2. For COBOL —

```
CALL "FDV$GET" USING fval,term,fidl,fidx].
CALL "FDV$PUTL" USING fval.
```

3. For PL/I —

```
CALL FDV$GET((fval,term,fidl,fidx]);
```

The argument abbreviations that are printed in lowercase letters stand for arguments that you must provide. They must be in the order shown for each call and must meet the functional requirements described in Table 5–5.

In the call descriptions in this manual, square brackets ([and]) enclose optional arguments. For example, in the FDV$GET call illustrated above, the argument for a field index (*fidx*) is required only to specify a particular field that is in an indexed field.

For calls that have more than one optional field, omitting one requires you to omit the others to its right. For example, if you omit the optional field name argument (*fid*) in the following call, you must also omit the field value argument (*fval*).

```
CALL FDV$PFT(term[,fid[,fval]])
CALL "FDV$PFT" USING term[,fid[,fval]].
```

For some calls, you can omit the entire list of arguments. For added clarity in this manual, these calls are listed both with and without the argument lists. For example, the full syntax of the FDV$PUTAL call is shown as follows:

```
CALL FDV$PUTAL(fval)
CALL FDV$PUTAL
```

The Form Driver's interface does not support null argument lists in calls. Using the second form of the FDV$PUTAL call above as an example, the following form is not recommended, for cross system compatibility, within an FMS application:

```
CALL FDV$PUTAL( )                    !The null argument form.
```

5.4.3 Status and Error Checking

As described fully in Section 6.25, the Form Driver includes a specific call, the FDV$STAT call, that returns status codes for the completion status of the last call that was processed. Table 5–1 lists the status codes and their meanings.

With all VAX languages, you can use the standard syntax for calling a function subprogram. The status value return syntax for the FDV$GET and FDV$PUTL calls is (*fncval* stands for the status value return):

Table 5-7 (Cont.): Listing of VAX-11 BASIC Form Driver Calls

Call Abbreviation	Summary and Forms
FDV$SPON	Turns on the Supervisor Only mode and prevents the operator from entering or changing data in fields to which the Supervisor-Only attribute was assigned in the Field Attributes Questionnaire (in the ASSIGN command in the Form Editor). The form is: CALL FDV$SPON
FDV$STAT	Returns the FMS status code for the last call that was processed as the value of the first argument. The value of the second argument is meaningful as an RMS system error code only if the value of the first argument is -4 (FDV$_IOL) or -18 (FDV$_IOR), indicating an error while trying to open or read a form library file. The form is: CALL FDV$STAT(*status%,stat2%*)
FDV$TERM	This is an optional call executable only in the VAX/VMS environment. The terminal channel number must be passed. Use SYS$ASSIGN to obtain the value. Further information can be found in the *VAX/VMS System Services Reference Manual*. If you do not use this call the default terminal channel is chosen. This call is helpful if you want to see form displays and system debugger statistics on a different terminal during program debug. The form is: CALL FDV$TERM(*chan*)

5.5.3 Building a VAX-11 BASIC Program

A VAX-11 FMS BASIC program is compiled as follows:

```
$ BASIC BASDEM [/LIST=BASDEM /OBJECT=BASDEM]
```

The application may be linked using one of three methods.

Method 1: Link with the Form Driver in STARLET.OLB (system object library)

```
$ LINK BASDEM
```

Method 2: Link with the Standalone Form Driver Library.

```
$ LINK BASDEM,SYS$LIBRARY:FDVL10/LIBRARY
```

Method 3: Link with the Form Driver shared library.

```
$ LINK BASDEM,FDVSHARE/OPT
```

The Interface for VAX-11 COBOL

In VAX-11 COBOL programs, only data names may be passed as arguments. All values that are passed must have been defined in the data division of the program. No string literals or numeric constants are allowed.

The COBOL interface assumes two specific data types when passing data to and from the Form Driver. These data types are:

for strings: left justified sign separate

for numbers: one longword computational

For example:

```
01 TERM PIC 9(9) COMP.
01 STAT PIC S9(9) COMP.
```

See Section 5.6.2 for a list of COBOL arguments that shows the necessary data types. You may create your own data structure provided that you use the listed data types in your Form Driver calls.

Since the FDV routines are *not* written in COBOL, special calling syntax must be used. String variables must be called with BY DESCRIPTOR prefix and computational variables with BY REFERENCE prefix. See Section 5.6.1.2 for detailed syntax for each call.

5.6.1 Using the Form Utility (FUT) to Create the Communication Structure for a COBOL Program

The Form Utility (FUT) creates the communication structure for a form used by a COBOL program. (See Chapter 3 on the Form Utility.) In response to the /CC option, FUT creates a COBOL library file. The output is a text file with the default file type .LIB.

At compile time, you request the library file by means of a COPY command in the data division of your program. (See the *VAX-11 COBOL Language Reference Manual* for details on the COPY command.)

Group items created by FUT have the same names as the field names in the form that you are using.

The library file contains the necessary communication structure. If you do not wish to use the structure provided, you may create your own.

The output of the Form Utility is in terminal format with respect to the COBOL program. If you want to use conventional format, you must reformat the file.

Chapter 3 includes an example of the VAX-11 COBOL structure that the Form Utility can produce.

5.6.2 Arguments for the COBOL Calls

Table 5-8 lists typical COBOL data types and data structures for each of the arguments in the Form Driver calls.

Table 5-8: Typical COBOL Data Types for Form Driver Arguments

Argument Abbreviation	Purpose, Data Type, and Picture Attributes
CHAN	Channel number; integer number. Picture: one longword computational, synchronized left. Passed by reference.

(continued on next page)

Table 5-8 (Cont.): Typical COBOL Data Types for Form Driver Arguments

Argument Abbreviation	Purpose, Data Type, and Picture Attributes
FID	Field name: up to 6-character string. Picture: any character, blank padded, left justified, sign separate, synchronized left. Passed by descriptor.
FIDX	Field and named data index: integer number. Picture: one longword computational, synchronized left. Passed by reference.
FLEN	Field length: integer number. Picture: one longword computational, synchronized left. Passed by reference.
FLNM	Form library file specification: a string whose length is the length of the file specification. Picture: any character, blank padded, left justified, sign separate, synchronized left. Passed by descriptor.
FNAME	Form name: up to 6-character string. Picture: any character, blank padded, left justified, sign separate, synchronized left. Passed by descriptor.
FVAL	Named data value, one or more field values, text for display on the bottom screen line. A string whose length is: • For a single field value, the length of the field. • For a string of field values, the sum of the lengths of all fields to be processed. • For a named data value, the length of the named data field maximum (60- actual length is variable. • For text to be displayed on the bottom line, the length of the text. Passed by descriptor.
IMPURE	Impure area: Binary array which should contain at least 8 (32-bit) integers. Passed by descriptor.
LINE	Starting line number for a displayed form: binary index. Picture: one longword computational, synchronized left. Passed by reference.
SIZE	The size of the impure area in bytes. To create an area large enough for your program's largest form, make the size 1000 bytes. Passed by reference. Picture: one longword signed computational, synchronized left. Passed by reference.
STATUS	Call completion status: integer number. Picture: one longword signed computational, synchronized left. Passed by reference.
STAT2	RMS system error code: integer number. Picture: one longword signed computational, synchronized left. Passed by reference.
TERM	Field terminator code: integer number. Picture: one longword computational, synchronized left. Passed by reference.

5.6.3 Syntax for the COBOL Calls

All of the COBOL Form Driver calls use the CALL statement. Table 5-9 summarizes the principal purposes and shows the full CALL statement syntax for each call. The arguments that you must supply are in lowercase letters, and optional arguments are enclosed in square brackets ([and]). The forms of calls that have no arguments are listed separately. The argument abbreviations and purposes are fully described in Table 5-8. See Section 5.1 for details on using FDV calls as function value returns.

Table 5-9: Listing of COBOL Form Driver Calls

Call Abbreviation	Summary and Forms
FDV$CLRSH	Clears the entire screen and displays the form with default field values. If a line number is specified, uses it as the starting line number for the form. The form is: `CALL "FDV$CLRSH" USING` *by descriptor fnam [by reference line].*
FDV$GCF	Returns the field name from the Form Driver argument list (and if the field is indexed, its index). The form is: `CALL "FDV$GCF" USING` *by descriptor fid [by reference fidx].*
FDV$GET	If a field name is specified, gets and returns the value for the field and its terminator. If a field name is not specified, synchronizes the program with the operator. The forms are: `CALL "FDV$GET" USING` *by descriptor fval by reference term by descriptor fid [by reference fidx].* `CALL "FDV$GET".`
FDV$GETAF	Gets and returns the value, field name (and, if the field is indexed, its index), and the field terminator for the field. The form is: `CALL "FDV$GETAF" USING` *by descriptor fval by reference term by descriptor fid [by reference fidx].*
FDV$GETAL	If the call includes an argument, gets and returns a concatenated string of all field values (and optionally the last field terminator used). If no arguments are specified, gets all values but only stores them in the impure area. The forms are: `CALL "FDV$GETAL" USING` *by descriptor fval [by reference term].* `CALL "FDV$GETAL".`
FDV$IDATA	Gets and returns the named data value that has the index specified. The form is: `CALL "FDV$IDATA" USING` *by reference fidx, by descriptor fval.*

(continued on next page)

Table 5-9 (Cont.): Listing of COBOL Form Driver Calls

Call Abbreviation	Summary and Forms
FDV$INIT	Supplies to the Form Driver the name of the impure area to use and its size. This call returns it own status code if the third argument is specified. The form is: CALL "FDV$INIT" USING *by descriptor impure, by reference size [,by reference status]*.
FDV$INLN	Gets and returns a concatenated string of the field values for the current line of the scrolled area that contains the specified field name and the last terminator used. The form is: CALL "FDV$INLN" USING *by descriptor fid,fval [,by reference term]*.
FDV$LCHAN	Supplies to the Form Driver the I/O channel to use for reading a form library file. The form is: CALL "FDV$LCHAN" USING *by reference chan*.
FDV$LCLOS	Closes the current form library file. The form is: CALL "FDV$LCLOS".
FDV$LEN	Returns the length of the specified field. The form is: CALL "FDV$LEN" USING *by reference flen, by descriptor fid, [,by reference fidx]*.
FDV$LOPEN	Opens the specified form library file. The form is: CALL "FDV$LOPEN" USING *by descriptor flnm*.
FDV$NDATA	Gets and returns the named data value that has the named data label specified. The form is: CALL "FDV$NDATA" USING *by descriptor fid,fval*.
FDV$OUTLN	Displays the specified string of field values in the current line of the scrolled area that contains the specified field. The form is: CALL "FDV$OUTLN" USING *by descriptor fid,fval*.
FDV$PFT	If the call includes an argument, processes the specified field terminator and identifies the appropriate field as the current field. (To find the current field name, use the FDV$GCF call.) If the specified terminator is a scrolled area terminator, the name of a field in the intended scrolled area must be specified, and if a string of values is specified, they will be displayed on the top or bottom line of the scrolled area after the terminator is processed. If no argument is included, the call processes the last terminator that was used. The forms are: CALL "FDV$PFT" USING *by reference term [,by descriptor fid,fval]*. CALL "FDV$PFT".

(continued on next page)

Table 5-9 (Cont.): Listing of COBOL Form Driver Calls

Call Abbreviation	Summary and Forms
FDV$PUT	Displays the specified value in the specified field. The form is: CALL "FDV$PUT" USING *by descriptor fval,fid [,by reference fidx]*.
FDV$PUTAL	Displays values in all fields of the form. If a concatenated string of values is supplied, each value must be the same length as the field in which it is to be displayed and the values must be in the same order that the FDV$GETAL call would produce for the form. Values from the string supplied are displayed in the first fields of the form, and defaults are displayed in any field that remain. If no string of values is supplied, default values are displayed in all fields. The forms are: CALL "FDV$PUTAL" USING *by descriptor fval*. CALL "FDV$PUTAL".
FDV$PUTL	If an argument is specified, displays the specified string on the bottom line of the screen. If no argument is specified, clears the bottom line. The forms are: CALL "FDV$PUTL" USING *by descriptor fval*. CALL "FDV$PUTL".
FDV$RETAL	Returns the current values for all fields in the form in the same order that the FGETAL call would produce. The form is: CALL "FDV$RETAL" USING *by descriptor fval*.
FDV$RETN	Returns the current value of the specified field. The form is: CALL "FDV$RETN" USING *by descriptor fval,fid [,by reference fidx]*.
FDV$SHOW	Clears the part of the screen that the specified form requires and displays the form with default field values. If a line number is specified, uses it as the starting line number for the form. The form is: CALL "FDV$SHOW" USING *by descriptor fnam [,by reference line]*.
FDV$SPOFF	Turns off the Supervisor Only mode and allows the operator to enter and change data in fields to which the Supervisor Only attribute was assigned with the Form Editor. The form is: CALL "FDV$SPOFF".
FDV$SPON	Turns on the Supervisor Only mode and prevents the operator from entering or changing data in fields to which the Supervisor-Only attribute was assigned in the Field Attributes Questionnaire (in the ASSIGN command in the Form Editor). The form is: CALL "FDV$SPON".
FDV$STAT	Returns the status code for the last call that was processed as the value of the first argument. The value of the second argument is meaningful as an RMS system error code only if the value of the first argument is -4 (FDV$__IOL) or -18 (FDV$__IOR), indicating an error while trying to open or read a form library file.

(continued on next page)

Table 5-9 (Cont.): Listing of COBOL Form Driver Calls

Call Abbreviation	Summary and Forms
	The form is: CALL "FDV$STAT" USING *by reference status,stat2*.
FDV$TERM	This is an optional call executable only in the VAX/VMS environment.
	The terminal channel number must be passed. Use SYS$ASSIGN to obtain the value. Further information can be found in the *VAX/VMS System Services Reference Manual*. If you do not use this call, the default terminal channel is chosen. This call is helpful if you want to see form displays and system debugger statistics on a different terminal during program debug.
	The form is: CALL "FDV$TERM" USING TERMINAL.

5.6.4 Building a VAX-11 COBOL Program

A VAX-11 COBOL program is compiled as follows:

```
$COBOL COBDEM /LIST=COBDEM /OBJECT=COBDEM)
```

The application may be linked using one of three methods.

Method 1: Link with the Form Driver in STARLET.OLB (system object library)

```
$LINK COBDEM
```

Method 2: Link with the standalone Form Driver Library.

```
$LINK COBDEM,SYS$LIBRARY:FDVLIB/LIBRARY
```

Method 3: Link with the Form Driver shared library.

```
$LINK COBDEM,FDVSHARE/OPT
```

5.7 The Interface for VAX-11 FORTRAN

FORTRAN can use a status value return to receive VAX/VMS return codes in the program.

Numeric arguments must be default longword integers. If you use real numbers or bytes, the calls do not work properly.

All subroutines may be called either as subprograms or as Status Value Returns. If they are called as functions, the name of the routine must be declared in an integer statement or an implicit integer statement (for example, IMPLICIT INTEGER (A-Z)). If called as functions, the subroutines may return the status of the call from the Form Driver on output.

It is recommended that VAX-11 FORTRAN applications which may migrate to PDP-11s not use the Status Value Return. The status returned is system dependent and will not be compatible across systems. To preserve the compatibility use the FMS FDV$STAT call to return a call's status.

5.7.1 Arguments for the FORTRAN Calls

Table 5-10 lists typical VAX-11 FORTRAN data types and data structures for each of the arguments in the Form Driver calls.

Table 5-10: Typical FORTRAN Data Types for Form Driver Arguments

Argument Abbreviation	Purpose, Data Type, and Data Structure
CHAN	Channel number: integer variable or constant. Passed as %REF.
FID	Field name: up to 6-byte string variable. Passed as %DESCR.
FIDX	Field and named data index: integer variable. Passed as %REF.
FLEN	Field length: integer variable. Passed as %REF.
FLNM	Form library file specification: string or constant (the size depends on application requirements and conventions). Passed as %DESCR.
FNAME	Form name: up to 6-byte string variable or constant. Passed as %DESCR.
FVAL	Named data value, one or more field values, text for display on the bottom screen line: string variable (the size depends on the application). Passed as %DESCR.
IMPURE	Impure area: byte array containing at least 8 (32 bit) integers. Passed as %DESCR.
LINE	Starting line number for a displayed form: integer variable or constant. Passed as %REF.
SIZE	The size of the impure area array in bytes. To create an area large enough for your program's largest form, make the size 1040 bytes. Passed as %REF.
STATUS	Call completion status: integer variable. Passed as %REF.
STAT2	RMS system error code: integer variable. Passed as %REF.
TERM	Field terminator code: integer variable. Passed as %REF.

5.7.2 Syntax for the FORTRAN Calls

All of the FORTRAN Form Driver calls use the CALL statement. The function subprogram form can also be used. Table 5-11 summarizes the principal purposes and shows the full CALL statement syntax for each call. The arguments that you must supply are in lowercase letters, and optional arguments are enclosed in square brackets ([and]). The forms of calls that have no arguments are listed separately. The argument abbreviations and purposes are fully described in Table 5-10.