Idea

Interactive

Data Entry/Access

Reference Manual

(AOS)

093-000151-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000151

Printed in the United States of America
Revision 01, December 1979

NOTICE

Idea
Interactive
Data Entry/Access
Reference Manual
(AOS)
093-000151

Revision History:

Original Release - August 1978
First Revision - December 1979 (Idea Rev. 3.00)

A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively, from revision 00.

Preface

This manual describes Data General's Interactive Data Entry and Access (Idea) system as it operates with the INFOS® file management system under the Advanced Operating System (AOS).

Prerequisite Knowledge

Before you read this manual you should understand both AOS and the INFOS system. We suggest that you read the following manuals:

- *Learning to Use Your Advanced Operating System* (069-000018)
- *INFOS® System User's Manual (AOS)* (093-000152)

System managers should also read the *AOS System Manager's Guide,* 093-000193.

If you plan to use RCX70 with Idea, you must read the *RCX70 Reference Manual AOS,* 093-000172.

Audience Definition

If you are a system manager, first read Chapters 1 and 2 for a basic understanding of Idea. Next, read Chapter 10, "How to Load and Generate Idea." This will tell you where to place Idea system files and local monitors, and what access privileges your programmers will need. You should also read Chapter 9 after you have determined your system's printing requirements. This chapter will show you how to set up Idea for the various printing formats.

If you are a programmer, you should read Chapters 1 through 6 before you begin writing programs.

Contents

Chapter 1 describes the capabilities of the Idea system running with the INFOS system under AOS. It shows you some of the different screen format types you can use with your programs, as well as some of the different INFOS file structures available.

Chapter 2 walks you through a programming example, from program design through the implementation steps to program execution.

Chapter 3 explains the Idea Format Generator (IFMT), the utility you use to create screen formats. This chapter describes the full set of IFMT field picture characters and the full set of attributes to assign to your screen data fields as well as how to design a 132-character format using the WIFMT utility.

Chapter 4 describes Idea's Field Processing Language (IFPL), which you use to write your programs.

Chapter 5 describes the process for using an INFOS data file with a program.

Chapter 6 describes the compilation process.

Chapter 7	is a reference section containing a detailed description of each IFPL command arranged in alphabetical order.
Chapter 8	lists the Idea system utilities.
Chapter 9	describes the printing options available with Idea.
Chapter 10	tells the system manager how and where to load Idea. It describes how to create global and local monitors, and how to invoke a local monitor, the initial process you need to run a program.
Appendix A	tells you how to convert RDOS Idea programs to AOS Idea programs, and vice versa.
Appendix B	describes the internal structure of the system COMMON file.
Appendix C	describes the internal structure of the system transaction file TRANS.
Appendix D	gives you listings of several application format/program modules. We provide the sources of these modules on the system tape.

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	Means
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\left\{ \begin{matrix} \text{required}_1 \\ \text{required}_2 \end{matrix} \right\}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
)	Press the NEW LINE or RETURN key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

) is the AOS CLI prompt.

Contacting Data General

If you:

- Have comments on this manual -- Please use the prepaid Remarks Form that appears after the Index.
- Require additional manuals -- Please contact your local Data General sales representative.
- Experience software problems -- Please notify your local Data General systems engineer.

End of Preface

Contents

Chapter 1 - Introduction to Idea

Chapter 2 - A Sample Programming Session

Chapter 3 - IFMT -- The Format Generator

Chapter 4 - The IFPL Language

Chapter 5 - Using INFOS Files with Idea Programs

Chapter 6 - Compiling the IFPL Program

Chapter 7 - IFPL Statements

Chapter 8 - Idea System Utilities

Chapter 9 - Printing

Chapter 10 - How to Load and Generate Idea

Appendix A - Converting Programs Between AOS and RDOS

Appendix B - The COMMON File

Appendix C - The Transaction File TRANS

Appendix D - Format/Program Module Listing

Tables

Illustrations

Chapter 1
Introduction to Idea

The Idea system is designed specifically for programs that display a format on the terminal screen as a guide for data input and output.

Screen Formats

The first step in writing an Idea program is designing the screen format. The format generator (IFMT) allows you to type on the screen as though you are typing on a blank piece of paper. You create data fields on the screen using COBOL-like picture characters -- 9s for numbers, As for letters, Xs for alphanumeric data, etc. These fields serve as windows through which you enter data into the program and the program displays data. You can position the cursor anywhere on the screen to type these fields.

You can also use any keyboard characters (except the exclamation point) as literals -- labels describing the data fields. For example, Figure 1-1 shows an accounts receivable screen format. The data fields appear brighter than the literal labels.

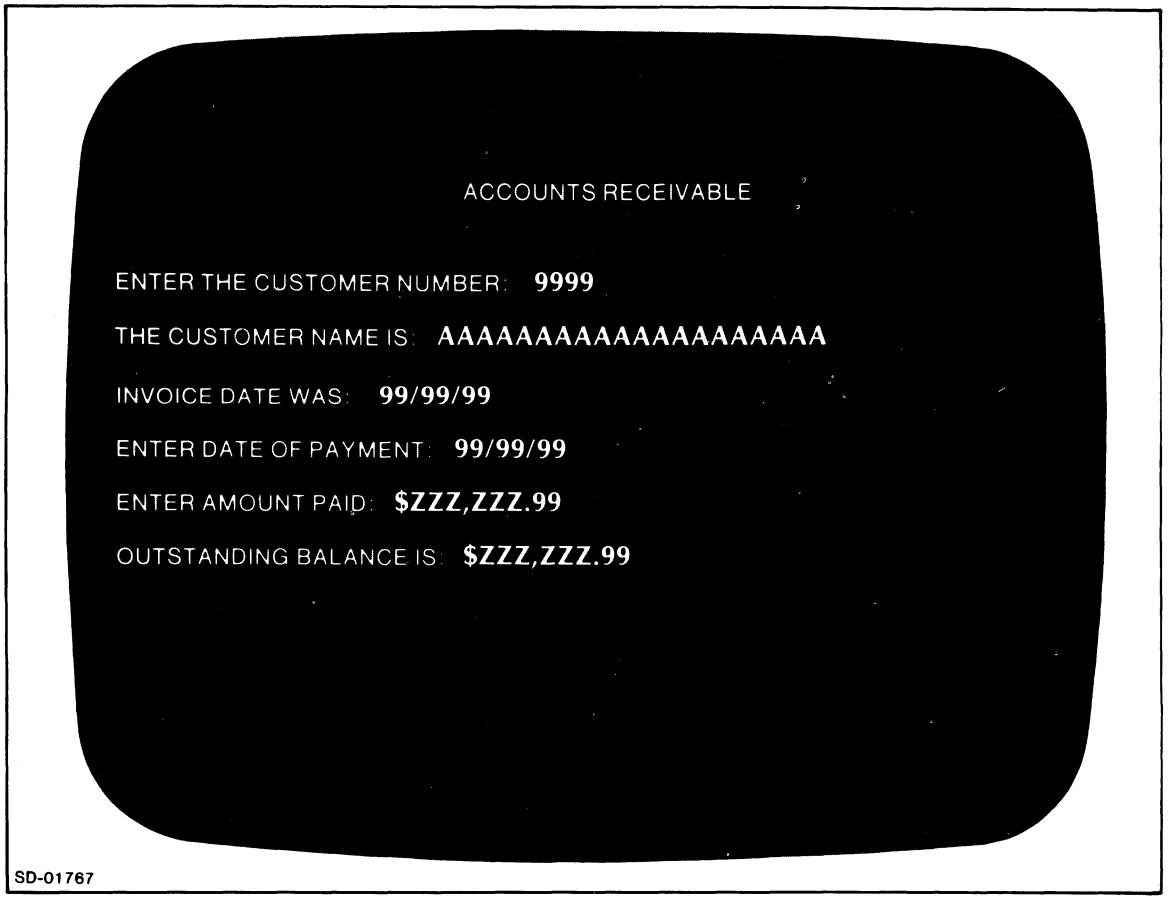


Figure 1-1. A Typical Screen Format as Defined with IFMT

Notice that the slashes in the dates are literal characters; each date is composed of 3 numeric fields. The Zs in the monetary fields are zero suppress characters; you may use them in place of 9s to eliminate leading zeros. We describe all the picture characters in Chapter 3.

Scroll Fields

Screens can also contain scroll areas. A *scroll area* is a series of lines that lets you repeat information. Figure 1-2 shows a screen with scrolled lines.

Figure 1-2. A Format with a Scroll Area

Attributes

After you have defined the fields and literals, you assign *attributes* to the fields. These attributes define how the program will use the field -- EDIT-only, DISPLAY-only, or both. They also allow you to control data input with additional options such as SECURE, which displays asterisks when an operator enters a value into a field.

IFPL Program

The screens are only half the story. Behind each screen may be a program written in Idea's Field Processing Language (IFPL). The IFPL programs contain PROCESS statements that connect the screen fields to routines in the program (see Figure 1-3).

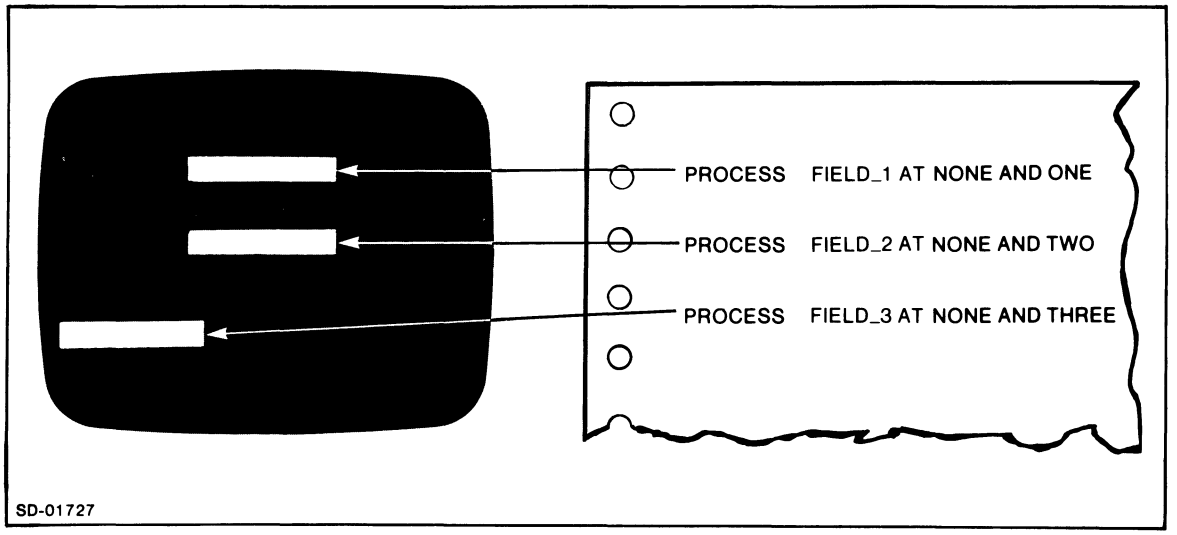


Figure 1-3. PROCESS Statements Connect Fields to Routines

Compiling and Executing a Format/Program

After you have defined the format and created the program source text, you compile the format and the program together to form an executable module using the SYNTAX command (described in Chapter 6). The compiler checks the screen field definitions and PROCESS statements for one-to-one correspondence. It reports any mismatches in DISPLAY/EDIT type, numeric/alphabetic/alphanumeric type, and so on. This type of error thus will not occur at runtime.

To execute a format/program module, you call up a *local monitor.* Your system manager will create the global and local monitors with the IDEASG command, described in Chapter 10. The global monitor is invisible; it operates behind the scenes, managing the system functions.

When you call up the local monitor, it asks for the name of the format you wish to use. When you give the format name, the monitor calls in the format/program module, displays the format on the screen, and waits to accept input.

In Chapter 2, we walk you through the above procedures, taking a programming session from problem design through its implementation to its format/program execution.

The File System

Idea uses AOS INFOS system DBAM files, which allow you several options for designing your database records and index structures. The options include the use of duplicate keys, approximate keys, generic keys, inverted keys, partial records, and subindexes. The *INFOS System User's Manual (AOS),* 93-000152, explains these options in detail.

To create a file, use the INFOS system ICREATE utility. You then define the file and records in a series of file definition statements within the program. Use file manipulation statements within the program to load a database, to access a file and its records, and to update a database. We explain this procedure in Chapter 5.

System Utilities

Table 1-1 shows AOS, INFOS, and Idea system utilities and tells where you can find information about each one.

Table 1-1. The System Utilities

AOS (See *Learning to Use AOS)*	
LINEDIT	A line-oriented text editor used to create program source text. (See *AOS LINEDIT Text Editor User's Manual,* 093-000218.)
SPEED	A character-oriented text editor, also used to create source text. (See *AOS SPEED Text Editor User's Manual,* 093-000197.)
INFOS (See *INFOS System User's Manual)*	
ICREATE	Creates data files (see Chapter 5) and the TRANS file (see Appendix C).
IDELETE	Deletes data files and the TRANS file.
Idea	
ALPHA	Allows you to define your alphabet. See Chapter 8.
CHGEM	Allows you to change error message and dialog files. See Chapter 8.
DEFCOM	Creates the COMMON file. See Chapter 8.
IDEASG	Generates global and local monitors. See Chapter 10.
IFMT	Creates screen formats. See Chapter 3.
ILIB	Creates a library of screen formats. See Chapter 8.
PALPH	Displays current set of alphabetic characters. See Chapter 8.
PFMT	Prints or displays information about screen formats. See Chapter 8.
PRINTF	Prints contents of printing buffer. See Chapter 8.
RDOSYNTAX	Compiles IFPL programs, producing RDOS-executable code. See Appendix A.
SYNTAX	Compiles screen format with program. See Chapter 6.
WIFMT	Creates wide (132 columns) print and hardcopy formats. See Chapter 3.

Templates

You receive two templates with the Idea documentation. Place these templates over the row of function keys above the keyboard and number pad.

The larger template is labeled IFMT on one side. Use the function keys labeled by this side when creating formats to enter FIELD, LITERAL, and ATTRIBUTE modes. These keys also help you move the cursor within the format, and they allow you to insert and delete lines and characters.

The other side of the larger template is labeled Idea INTERACTIVE DATA ENTRY AND ACCESS. Operators use the keys labeled by this template and by the smaller template when entering data into a screen format.

We explain the IFMT function keys in Chapter 3 and the operator function keys in Chapter 10.

End of Chapter

Chapter 2
A Sample Programming Session

This chapter introduces you to the basic Idea utilities by taking you through a sample programming session. Please follow along with the example as we create and run a simple Idea screen format/program module.

To create and run a program, follow these steps:

1. Define the screen format using IFMT.
2. Write the program source text using one of the AOS text editors.
3. Compile the format and the program together using the SYNTAX utility.
4. Run the program using the local monitor (see Chapter 10).

Problem Definition

We will create a simple Idea format/program to balance a checkbook. The program will accept as input a starting balance, a deposit, and a withdrawal. It will then add the deposit to the starting balance, subtract the withdrawal, and display the new balance on the screen.

This program will not use a data file, because it does not store any information.

Defining the Screen Format

Place the larger template with the side labeled IDEA IFMT over the row of function keys.

To call IFMT, give this command from the CLI:

```
IFMT)
```

IFMT will respond:

*NEXT FORMAT:*_ _ _ _ _ _ _ _

You answer by typing the name of the format, CHECKBOOK, followed by NEW LINE:

NEXT FORMAT: CHECKBOOK)

IFMT will then ask you for a format type:

*TYPE(H OR P OR NONE)*_

Respond by striking the NEW LINE key to answer NONE. (H and P refer to printing formats; we'll explain them in Chapter 3.)

Defining the Screen Literals

When you first create a screen format, IFMT places you in LITERAL mode and displays a reminder, MODE:LITERAL, in the lower right-hand corner of the screen, as in Figure 2-1.

In LITERAL mode, you can move the cursor anywhere on the screen to type out descriptive or instructional information, using any of the graphic keyboard characters (except the exclamation point).

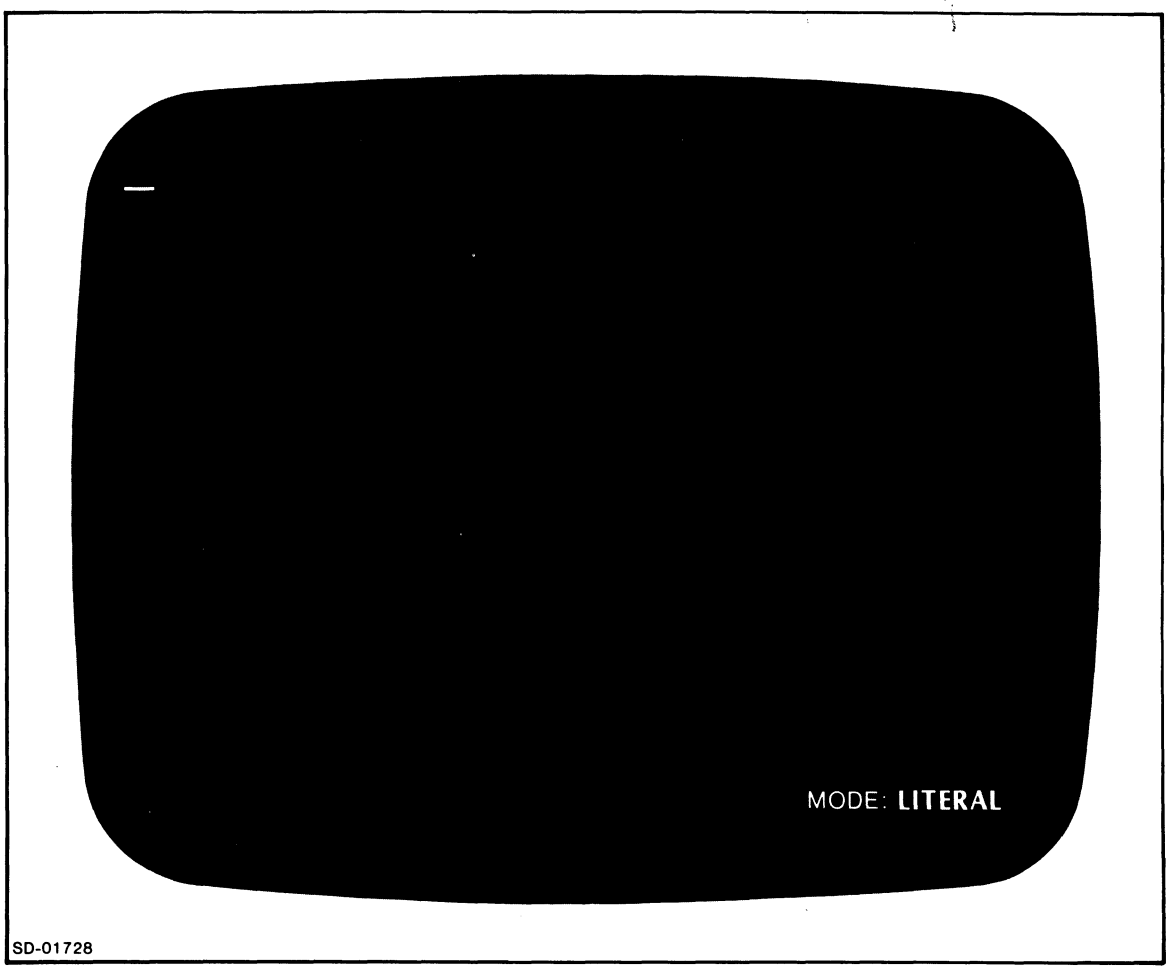


Figure 2-1. The Initial Screen

The literals don't interact with the program in any way; they are simply labels that you place on the screen to help operators use the format.

Figure 2-2 shows the literals to type for the CHECKBOOK screen. Just move the cursor to the desired location with the cursor arrow keys, and type the literals using the terminal keyboard as you would a typewriter keyboard.

Figure 2-2. The Literals for CHECKBOOK

Defining the Data Fields

To define data fields in a format, place IFMT in FIELD mode by striking the SHIFT and FIELD keys. IFMT will then display MODE FIELD in the lower right-hand corner of the screen.

You can shift back and forth between FIELD and LITERAL mode by striking the LITERAL/FIELD key. IFMT will always display a reminder in the lower right-hand corner about which mode you are in.

The CHECKBOOK format requires five data fields. The first four are numeric fields. To reserve a place for a number, type 9 in FIELD mode. To use a decimal point in a numeric field, you type a period (.) in the place you want the decimal point. For example, to create a numeric field with four integer places, a decimal point following them, and two decimal fraction places (representing cents in this example), you would type

9999.99

The fifth field in our example will accept any keyboard character as input, so type one X. Xs signify alphanumeric data.

Now, in FIELD mode, use the cursor control keys to position to the desired locations, and define the data fields so that your format looks like Figure 2-3.

Figure 2-3. Literal and Data Field Information for CHECKBOOK

Assigning Attributes

After you've created the labels and defined the screen fields, you assign attributes to the fields. To begin this process, press the shift key and strike the ATTRIBUTE key.

After verifying the legality of the field definitions, IFMT displays flashing question marks in place of the first field's picture characters. It also positions you to a series of attribute questions about this field at the bottom of the screen (see Figure 2-4). To assign an attribute, type the letter Y after the attribute.

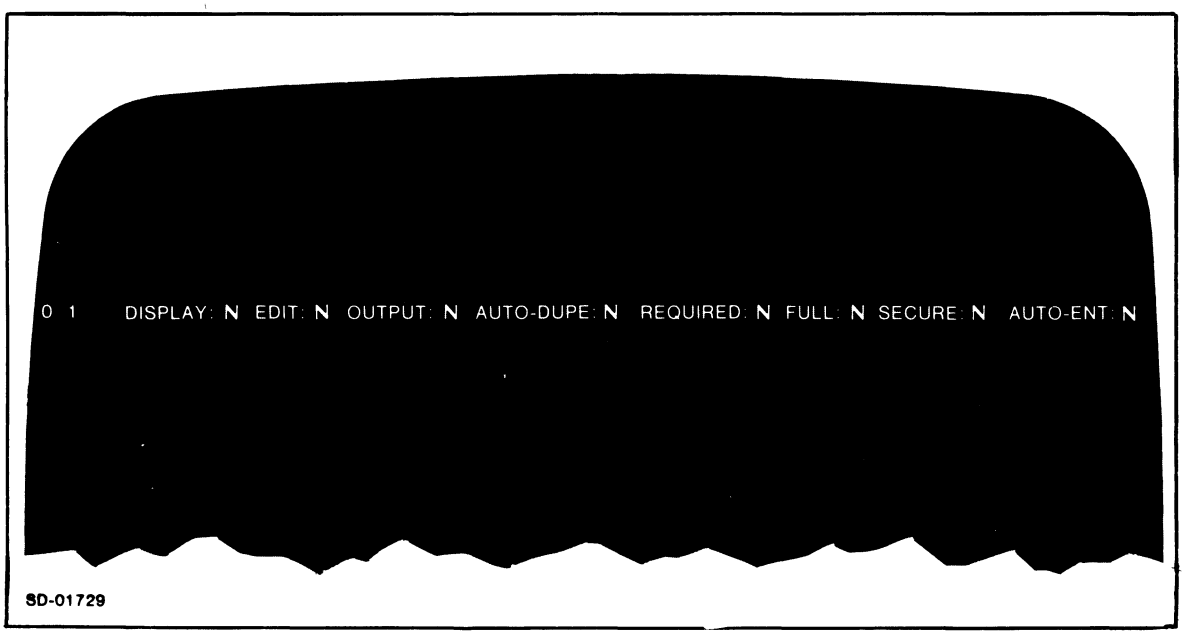


Figure 2-4. The Attribute Query Line

When you create a new format, its field attributes are all set to N for NO. To leave an attribute as it is, strike only the NEW LINE key. To change N to Y, type Y and NEW LINE; to change Y to N, type N and NEW LINE.

You will want to enter data into your program through the first field, labeled ENTER THE PREVIOUS BALANCE, so give it the EDIT attribute. To do this, skip the DISPLAY attribute by striking NEW LINE at that position, thus moving to the EDIT attribute. You then type Y in place of N (see Figure 2-5).

Notice the numerals 01 at the beginning of the attribute line. This tells you that you are at field #1.

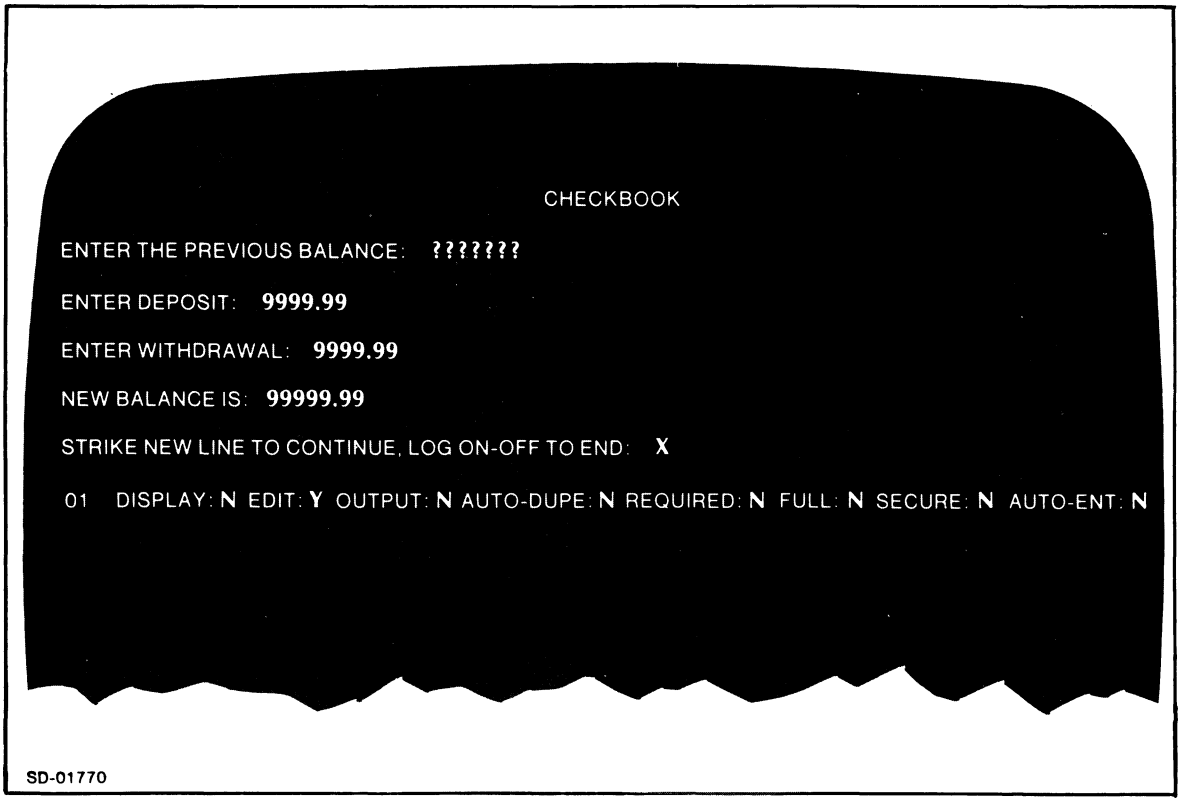


Figure 2-5. The CHECKBOOK Screen: Assigning the EDIT Attribute to the First Field

After you assign the EDIT attribute, strike the NEW LINE key for the rest of the attributes; they are optional. When you've completed the attribute line for the first field, IFMT will display hyphens in place of that field's picture characters. Then it will flash question marks in the next field, and display a new set of attribute choices for the second field (see Figure 2-6).

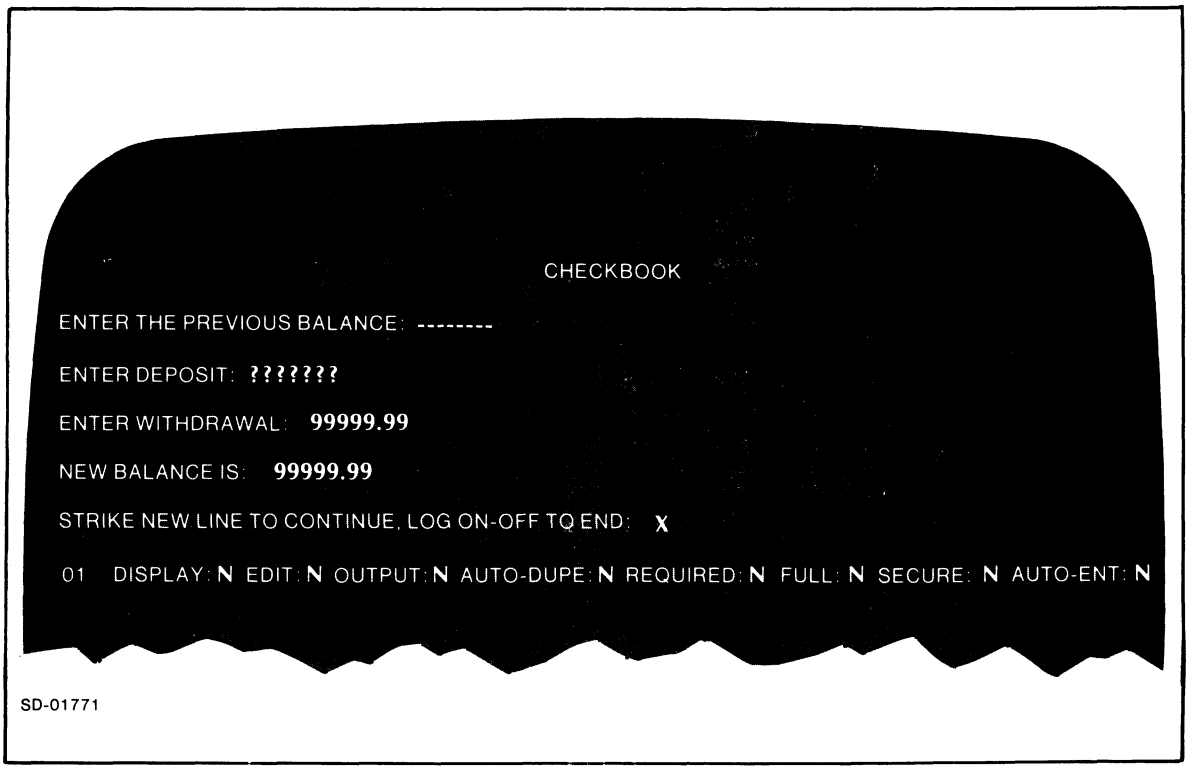


Figure 2-6. After You've Assigned Attributes to a Field, IFMT Asks About the Next One

Assign the EDIT attribute to the second and third fields. Next, assign the DISPLAY attribute to the fourth field, labeled NEW BALANCE IS, and the EDIT attribute to the fifth. After you finish, IFMT asks if you want to link to another format, and underlines a space for you to enter the other format's name. This feature allows you to link the current format to itself so it will run repeatedly, or to link it to another format/program module that will run after the current one is complete.

We do not want to link the format, so we enter just NEW LINE as in Figure 2-7.

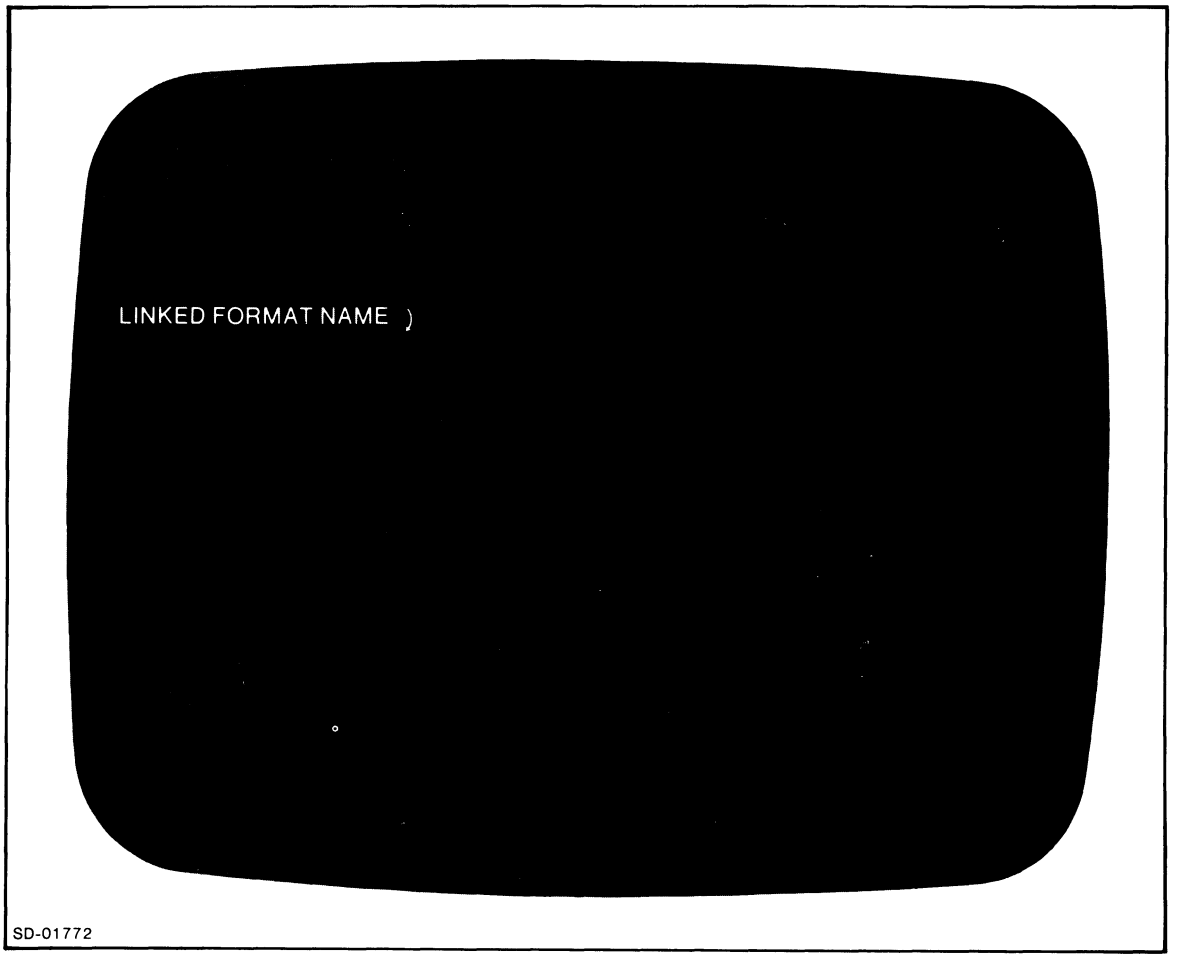


Figure 2-7. IFMT Format Link Option

Next, IFMT displays the message in Figure 2-8.

After IFMT compiles the format, it warns you that the format is not associated with any program. Then it asks you to specify another format to create or modify.

The example format is now complete, so you can strike NEW LINE. Finally, IFMT returns you to the CLI, and you're ready to write the program.

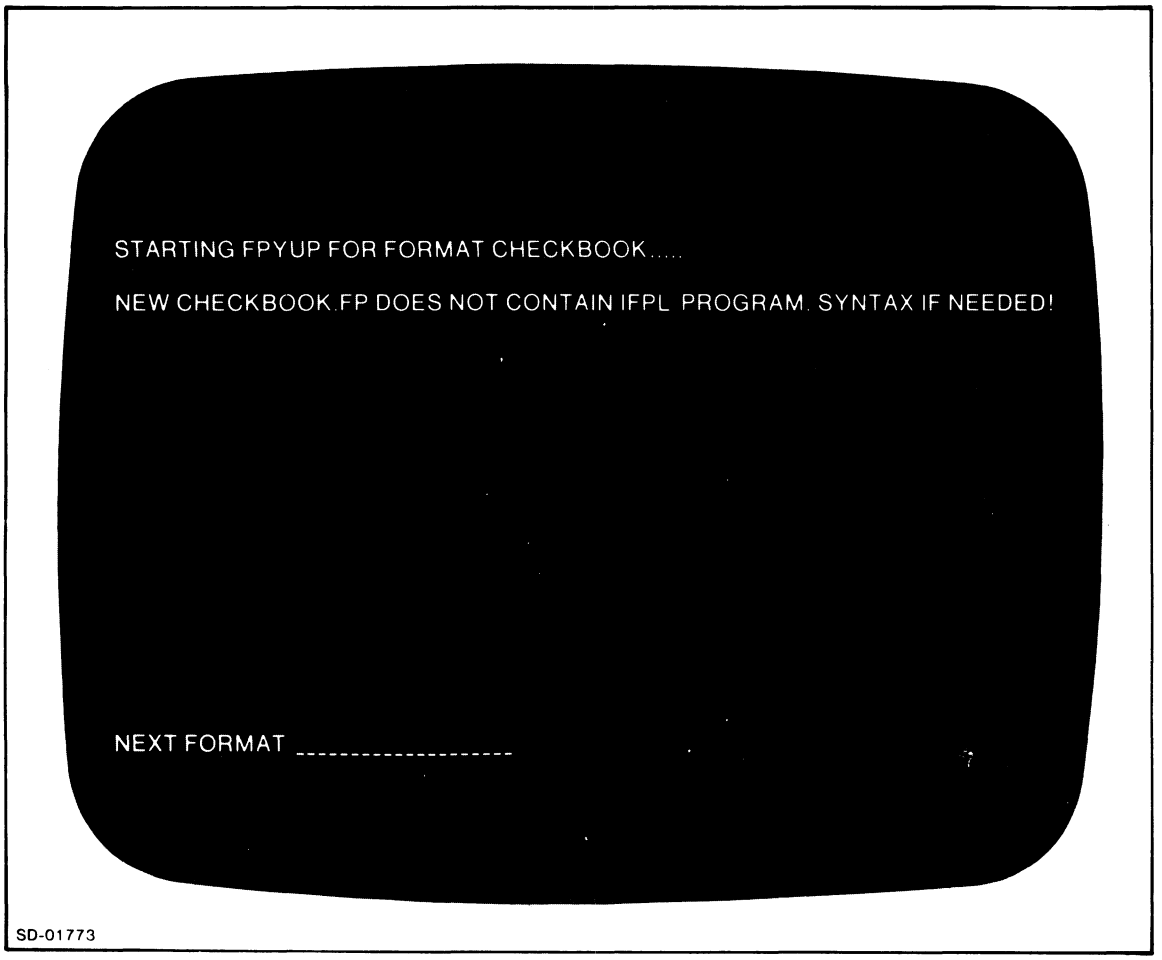


Figure 2-8. IFMT Puts the New Format Through a Special Program to Create an Idea-readable .FP File

Writing the Program

The sample program in Figure 2-9 will accept a balance, a deposit, and a withdrawal as input from the format. It will then perform some arithmetic and display a new balance. It will also halt until we strike any keyboard character; at this point the program will erase the screen and display a fresh format.

The program consists of routines that perform these tasks and PROCESS statements that connect the screen fields to the routines (see Figure 2-9).

```
NAME CHECKBOOK

PROCESS  BALANCE AT NONE AND GETBALANCE
PROCESS  DEPOSITS AT NONE AND GETDEPOSIT
PROCESS  WITHDRAWALS AT NONE AND GETWITH
PROCESS  NEWBALANCE AT CALCBALANCE AND NONE
PROCESS  FILLER AT NONE AND REPEAT

GETBALANCE:
        STORE BALANCE
        RETURN

GETDEPOSIT:
        STORE DEPOSITS
        RETURN

GETWITH:
        STORE WITHDRAWALS
        RETURN

CALCBALANCE:
        ADD DEPOSITS BALANCE NEWBALANCE
        SUBTRACT WITHDRAWALS NEWBALANCE NEWBALANCE
        DISPLAY NEWBALANCE
        RETURN

REPEAT:
        RETURN 1

FINISH
```

Figure 2-9. The Source Text of Our Program

Each PROCESS statement in Figure 2-9 contains the keyword NONE. A PROCESS statement for an EDIT field contains the phrase,

AT NONE AND routinename

A PROCESS statement for a DISPLAY field includes the phrase,

AT routinename AND NONE

You can give a field both the EDIT and DISPLAY attributes, in which case the PROCESS statement will contain the phrase,

AT routinename1 AND routinename2

Note that if the screen data fields have the EDIT and/or DISPLAY attributes, the fields must correspond exactly to the PROCESS statements. When you run the program, the monitor matches the first field with the first PROCESS statement, the second field with the second PROCESS statement, and so forth.

Furthermore, you must group the PROCESS statements together with no other statements between them.

When you run this program, the monitor will wait for you to type a value in the first field. The monitor will then retrieve this value and pass control to the program routine that is identified by the tag in the first PROCESS statement. At this routine, labeled GETBALANCE, the program copies the value in the variable BALANCE and returns control to the monitor. The monitor then repeats this process for the variables DEPOSITS and WITHDRAWALS.

The statement

```
PROCESS NEWBALANCE AT CALCBALANCE AND NONE
```

sends program execution to the routine named CALCBALANCE. Since this routine uses a DISPLAY field, the monitor passes control directly to the program without waiting for operator input. The ADD statement adds the values of DEPOSIT and BALANCE and places the result in the variable NEWBALANCE. The SUBSTRACT statement subtracts the value of WITHDRAWALS from NEWBALANCE and places the result in NEWBALANCE. The DISPLAY statement displays the result on the screen in the field with the DISPLAY attribute.

The statement

```
PROCESS FILLER AT NONE AND REPEAT
```

along with the routine labeled REPEAT, simply delays the end of the program until you type an alphanumeric character. Without this PROCESS statement and routine, the monitor will clear the screen immediately after displaying NEWBALANCE; it will then request another format name.

To run the program again, strike the NEW LINE key. To stop the program, press the SHIFT key and strike the LOG ON-OFF function key (function key 1).

Creating Source Text

To create source text for your programs, use one of the AOS text editors, SPEED or LINEDIT. If you name program files formatname.UP, you can use a simple version of the SYNTAX command to compile the format and the program. In this example, use CHECKBOOK.UP as the program filename.

Compiling CHECKBOOK

To compile your program, give this command from the CLI:

```
SYNTAX CHECKBOOK)
```

Executing the Program

To execute a format/program module, you must first call up the local monitor, which your system manager created with the IDEASG command (described in Chapter 10). The default local monitor name in LIDEA. If the system manager used the default names, you call the local monitor from the CLI by typing:

```
X LIDEA)
```

The monitor will ask for your password. This is optional; you can just type NEW LINE. Then it asks for the name of the format you wish to use. After you supply this, the monitor asks if you want the system to tell you the length and data type of each EDIT field. Type Y for yes, NEW LINE for no.

When you've completed the log-on sequence, the monitor displays the format on the screen and waits for your input to the EDIT fields.

End of Chapter

Chapter 3
IFMT--The Format Generator

This chapter describes the Idea Format Generator, IFMT, which you use to create and modify screen, print, and hardcopy formats. It also describes the Wide Format Generator, WIFMT, which you use to create and modify wide formats (up to 132 characters across) for output on a line printer or hardcopy device.

A format consists of the following:

Literals	These serve as headings, labels, and dividers for data fields.
Data fields	These are pictures of your program variables that set the variable's format location, format appearance, and data type (numeric, alphabetic, or alphanumeric).
Attributes	These define the usage of the data fields.
Scroll areas	These are areas in which you roll lines of data.
Partial screens	These are areas from one format that you overlay onto another format as a literal.

Entering IFMT

To enter IFMT from the CLI, type this command:

```
IFMT)
```

IFMT asks for the name of the next format. You may supply a new format name or the name of an existing format. If you are modifying an existing format, the name can be a pathname up to 24 characters long. The filename portion of the pathname must be 10 or fewer characters if you will link to this format via another format.

If you specify a pathname for an existing format, IFMT will rewrite the format to that pathname directory. But if you specify an existing format without a pathname, IFMT will retrieve it via the SEARCHLIST and rewrite it to the working directory.

If the ACL settings limit your file access, or if the format file is currently open, or if the pathname contains an illegal character, you will get this error message:

NAME, ACL, OR IN-USE ERROR

This will occur after you answer the next question, format TYPE.

After you give the format name followed by NEW LINE, IFMT asks for the format type:

TYPE(H OR P OR NONE)

You can use an IFMT format in one of three ways: in normal Idea monitor operation on a 6053 terminal; to produce formatted line printer output; and in conjunction with a DASHER™ printing terminal. Depending on how you want to use the format, enter one of these responses:

NEW LINE Create a screen format for normal Idea operation on a 6053 terminal. Format length may be up to 23 lines, the number of lines on the terminal screen minus one line for messages (line 24).

P NEW LINE Create a format for line printer operation with the PRINTF utility (described in Chapter 8). This mode allows formats up to 80 lines long. It also allows you to use the PREV PAGE and NEXT PAGE keys on the IFMT side of the large template to move around within the format. It disables the questions about field attributes, but asks you how long the print format will be.

H NEW LINE Create a format for interactive use with a DASHER printing terminal. As with the P response, this mode allows formats up to 80 lines long, and lets you use the PREV PAGE and NEXT PAGE keys.

IFMT Commands

Table 3-1 lists the IFMT commands. Use these commands when creating formats. Remember to place the large template with the side labeled Idea IFMT over the function keys.

You may escape from an IFMT session any time before you enter ATTRIBUTE mode by striking the ESC key. IFMT will display the message

INTENTIONAL SCREEN ABORT

and will return to the NEXT FORMAT question. If you were editing an existing format, the format files will remain as they were before you began altering them. If you were creating a new format, it will exist but will contain nothing.

Table 3-1. IFMT Command Repertoire (6053 Terminal)

Command	Function
DEL	Substitutes space for character to left of cursor.
DELETE CHAR	Deletes character at cursor screen location and shifts remaining characters on the same line left one position.
INSERT CHARS	Commences insert mode operation. Inserts characters you type at cursor. Shifts to the right the remaining characters on the same line. Deletes the last character on the line. You can cancel insert mode by a second INSERT CHARS or by vertical cursor movement.
DELETE LINE	Deletes line at cursor screen location and moves remaining lines up one line.
INSERT LINE	Opens line at cursor screen location and moves lower lines down one line. Last line is deleted.
FIELD	Puts IFMT in FIELD mode.

Table 3-1. IFMT Command Repertoire (6053 Terminal) (continued)

Command	Function
LITERAL	Puts IFMT in LITERAL mode.
PRINT (Cursor Pad)	Prints screen format on line printer.
ATTRIB	Indicates to IFMT that format is complete. IFMT responds by displaying attribute questions for each format field.
BACK TAB (unmarked key on cursor pad)	Moves cursor back one field at a time while in ATTRIB mode. Use it if you answer the field attribute question incorrectly.
Cursor Controls	Position cursor at any point on the screen.
Printer Format Commands (used with both P- and H-type formats)	
Command	**Function**
NEXT PAGE	Displays next 20-line page.
PREV PAGE	Displays previous 20-line page.
Special Format Characters	
Character	**Function**
@ (Field Mode only)	First @ used indicates start of scroll area. Second @ used ends the scroll area. You may use this sequence repeatedly.
!	Partial screen delimiter. A pair of exclamation points brackets a partial screen area.
//FF//	Form Feed. When used in a printing format, PRINTF will replace it with a form feed.
//HEADING//	For repeated literals in formats used with PRINTF. Use for current PAGE heading only and current scroll heading if any. PRINTF will reproduce only "last seen" headings when it encounters a form feed.

Literals and LITERAL Mode

When you create a new format, IFMT places you in LITERAL mode. In this mode, you can use any keyboard character (except the exclamation point) to create headings, labels, and dividers. Literals don't interact with programs; they serve only as labels.

With a 6053 format, the monitor displays the literals as they appear when you create them. With a P type format, the PRINTF utility reproduces the literals. With an H type format, the monitor does not display them.

To change from LITERAL to FIELD mode, strike the SHIFT and FIELD keys. To change from FIELD to LITERAL mode, strike only the LITERAL key.

Data Fields and FIELD Mode

Once you're in FIELD mode, use the following characters to create the data field pictures.

Character	Definition
A	Alphabetic character
9	Numeric character
X	Alphanumeric character
.	Decimal point
Z	Zero suppress character
+	Signed field character
$	Floating currency symbol
*	Check protection character
,	Numeric field comma

NOTE: All characters but the A and X are numeric field designators.

You cannot mix Xs, As, and 9s when creating data field pictures. IFMT sees a data field as an unbroken string of similar characters. Therefore, AAAA is a single data field, but AA99 defines two data fields (one alphabetic, one numeric), and AAXX99 defines three data fields. Also, do not space within a string. For example, XXXX defines one data field, but XX XX defines two.

The characters that delimit data fields are:

- Space
- End of Line
- Literal Character
- Dissimilar Field Designator

Alphabetic Fields

The picture character A defines a character position as alphabetic. For example, if you define a field as AAAAA, it will accept up to five alphabetic characters.

The Idea system file ALPHABET.TB defines the set of alphabetic characters. To change this file, use the ALPHA utility described in Chapter 9. If you don't change them, the legal alphabetic characters are the letters A-Z and the space.

Alphanumeric Fields

The picture character X defines a character position as alphanumeric. You may enter any graphic keyboard character in an alphanumeric field.

Numeric Fields

The picture character 9 defines a character position as numeric. For example, if you define a field as 99, it will accept any two digits (0-9). If you define a field using only 9s, you can't enter a decimal point; if you try to, the system will issue an error message.

Decimal Point

Define the field position of a numeric value's decimal point by placing a decimal point in the desired location of the field's picture. When you enter data into the field, you must explicitly enter the decimal point for values with decimal fractions. If you don't enter it explicitly, the system assumes that the value is an integer.

Zero Suppress Character

To suppress leading zeros, place the Z character in the places where you don't want leading zeros to appear. For example, instead of a numeric picture 9999.99, you could specify ZZZZ.99.

Signed Field Character

To display a signed value (i.e., + or -), you use the sign character (+) in the field picture. You can place a single + in the rightmost character position of the picture, such as 9999.99+. On output, the system will display the sign character on the right; for example, 1332.50+ or 0001.00-.

You can also place the + to the left of the numeric picture characters, such as +9999.99. With such a picture, the monitor will display the sign on the left but will not suppress leading zeros; i.e., -0005.72, or +0423.00. To suppress leading zeros, use multiple + signs, such as ++++9.99. On output, the monitor will suppress leading zeros and place one sign immediately preceding the numeric value; for example, +83.45, or -4729.25.

Currency Symbol

Placing a single dollar sign at the left of a data field picture will display one currency character in that position; placing a series of dollar signs there will suppress leading zeros and display one currency symbol. For example, a picture of $99.99 and an entry of 5.43 results in $05.43. A picture of $$$.99 and an entry of 5.43 results in $5.43.

Check Protection

The asterisk picture character replaces a leading zero with an asterisk. It is not a floating character, so to suppress all leading zeros, use a picture that consists of all asterisks to the left of the decimal point, such as ******.99.

Comma

Use the comma in field pictures according to its American usage. It will appear on output only when it's necessary. For example, with a picture $$,$$$.99, an entry of 2000 results in a display of $2,000.00. An entry of 431.50 results in a display of $431.50. Do not enter the comma explicitly. For example, type 2000 in a field, not 2,000.

Restrictions

You can use the following combinations of characters only if you observe certain restrictions.

The Floating Currency and Sign Characters

If you use the dollar sign with the sign character, you can use only one of them as a floating character. Specify the floating character by typing it at least twice. Place the other character outside the floating one; it becomes fixed in that position.

Examples

+$.99 Both are fixed.

+$$.99 The dollar sign floats; the sign is fixed.

$++.99 The sign floats; the dollar sign is fixed.

WARNING: If you use these characters together, note that you must reserve space for the digits; the + and $ characters each take up one character position. Therefore, the picture +$.99 will only allow you to display decimal fractions; it has no spaces for digits to the left of the decimal point. The pictures $++.99 and +$$.99 can only display one digit to the left of the decimal point, such as $-3.49, or +$2.50. They will both suppress leading zeros, such as $+.50, or -$.37.

The Zero Suppress and Check Protection Character

You cannot use the Z and the * together.

Other Combinations

When you use the dollar sign or the signed field character with the zero suppress or the check protection character, you can only use one dollar sign or sign character in the leftmost position. The dollar sign or sign character is fixed in that position.

Also, you can't place a $ character to the right of a decimal point.

Fields During Program Execution

During program execution, the system steps through the fields in the order in which they appear on the screen. It moves from left to right and from top to bottom, unless the program specifically calls for another order. At each field with the DISPLAY or EDIT attribute, the system pauses to execute the program routine associated with that field.

Page and Scroll Mode

Up to now, we have used only page mode formats. During program execution, fields defined in page mode appear only in the positions specified during format creation. Scroll mode fields, however, allow you to display multiple lines of fields.

To specify scrolled fields, you strike the commercial at (@) key while in FIELD mode; this begins the scroll area. The first line of the scroll area will be the line containing the fields that follows the @. A second @ ends the scroll area, returning you to page mode. You place the fields that you wish to repeat on succeeding lines between the @ signs. For example, Figure 3-1 shows a typical scroll specification, which contains two numeric fields and a three-line scroll area.

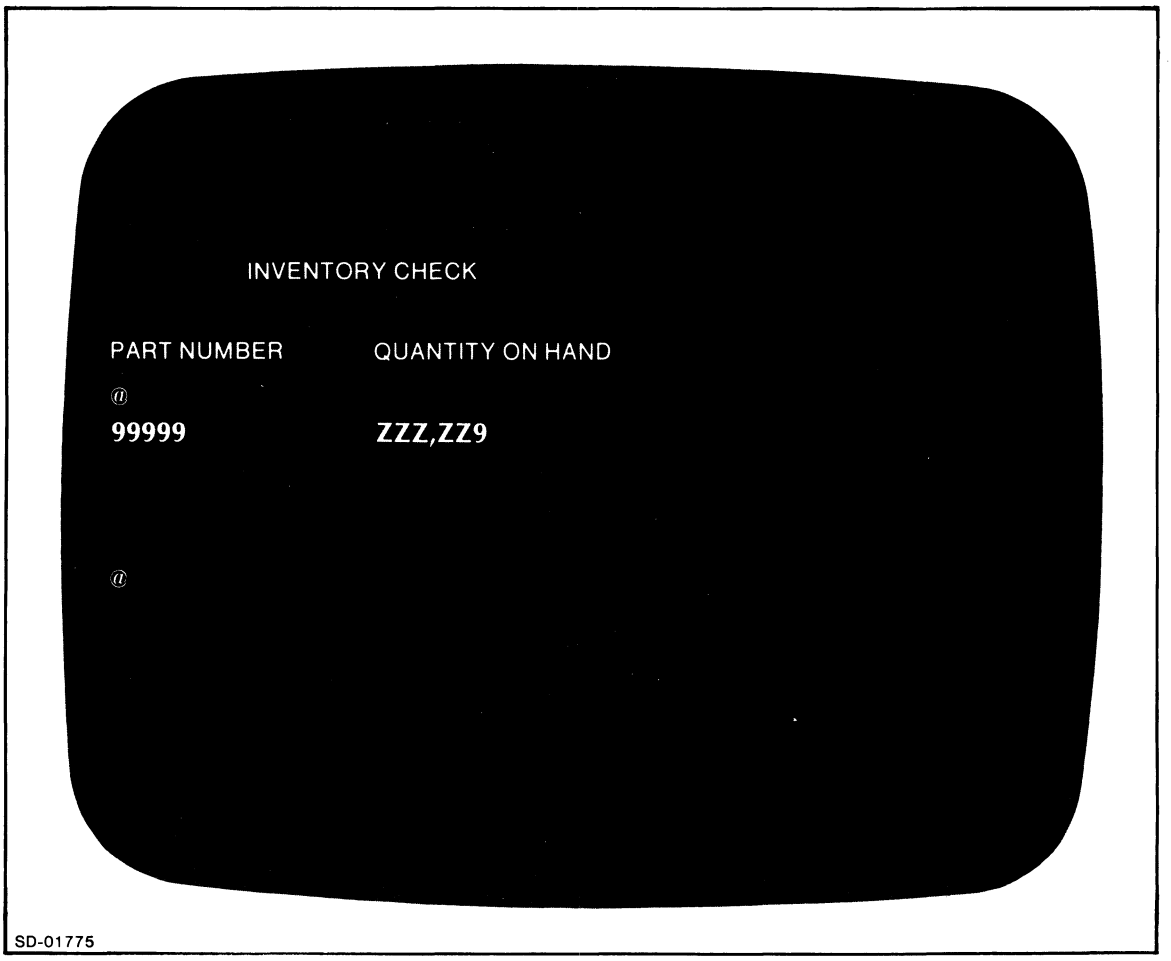


Figure 3-1. A Scroll Field Specification

Figure 3-2 shows an operator's console screen during program execution. The program could call the scroll lines of output to the screen in several ways. It could display the information automatically, or do it line by line, triggered by an operator entry such as a part number.

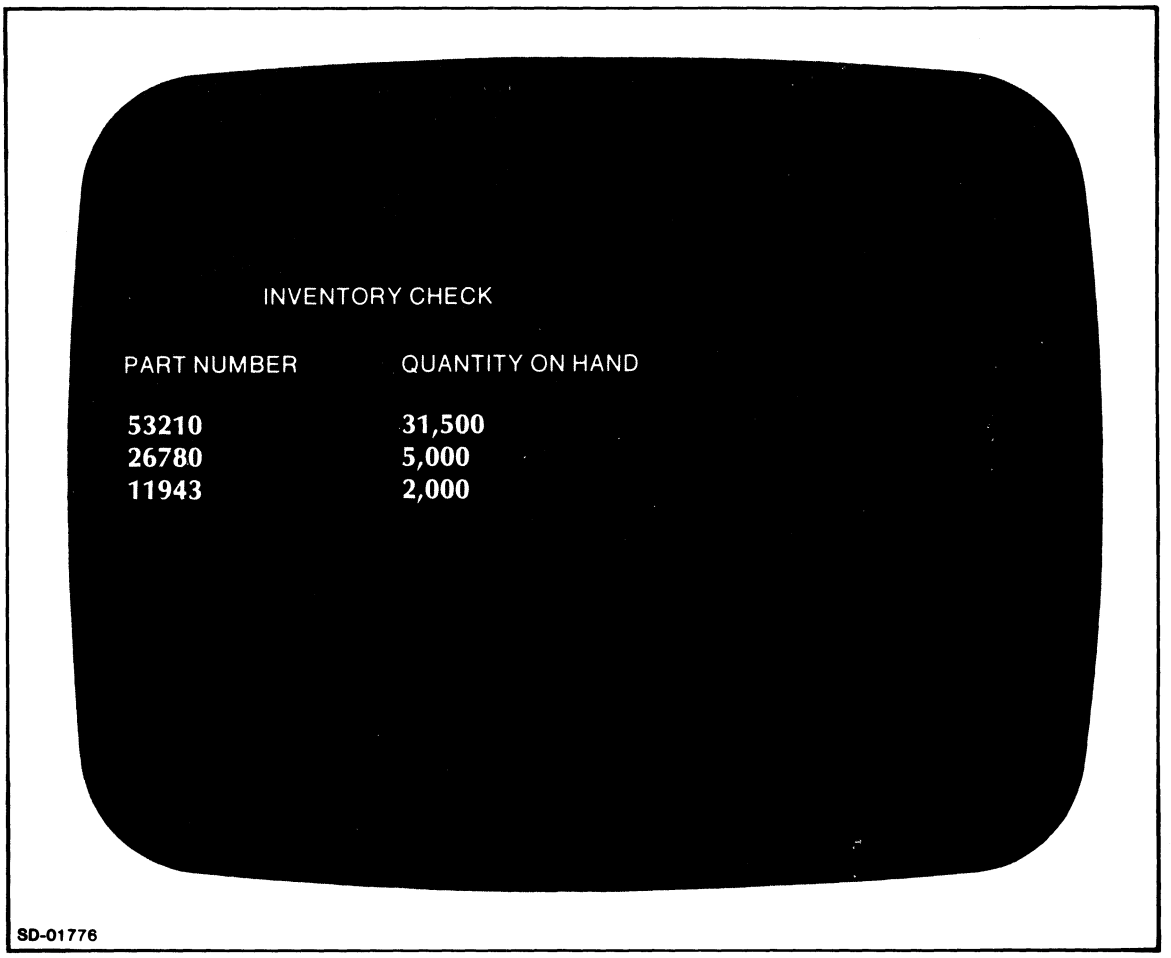


Figure 3-2. The Displayed Scroll Fields

Overlaying Partial Screens

In normal operation, the monitor erases an entire format from the screen prior to displaying a new one. You can retain areas of one format and display them with another format by using partial screens. You will normally use partial screens for operator reference. Data left from a previous screen has the status of a literal; i.e., you can't change it.

To overlay an area from one format onto another, you enclose the corresponding area of the second format in exclamation points (!) in LITERAL mode. When the monitor calls the second format, it will erase only the portion of the first format that corresponds to the area of the new format within the exclamation points. It will continue to display the rest of the first format.

For example, Figure 3-3 shows two formats. The second one contains an area marked off by exclamation points. Figure 3-4 shows what the screen will look like when the monitor loads the second format.

In Figure 3-4, the monitor substitutes the CURRENT CHARGES portion of the second format for the ADDRESS, CITY, and STATE portion of the first format. It leaves the company name and the customer name on the screen as a literal.

A format may contain any number of overlay areas.

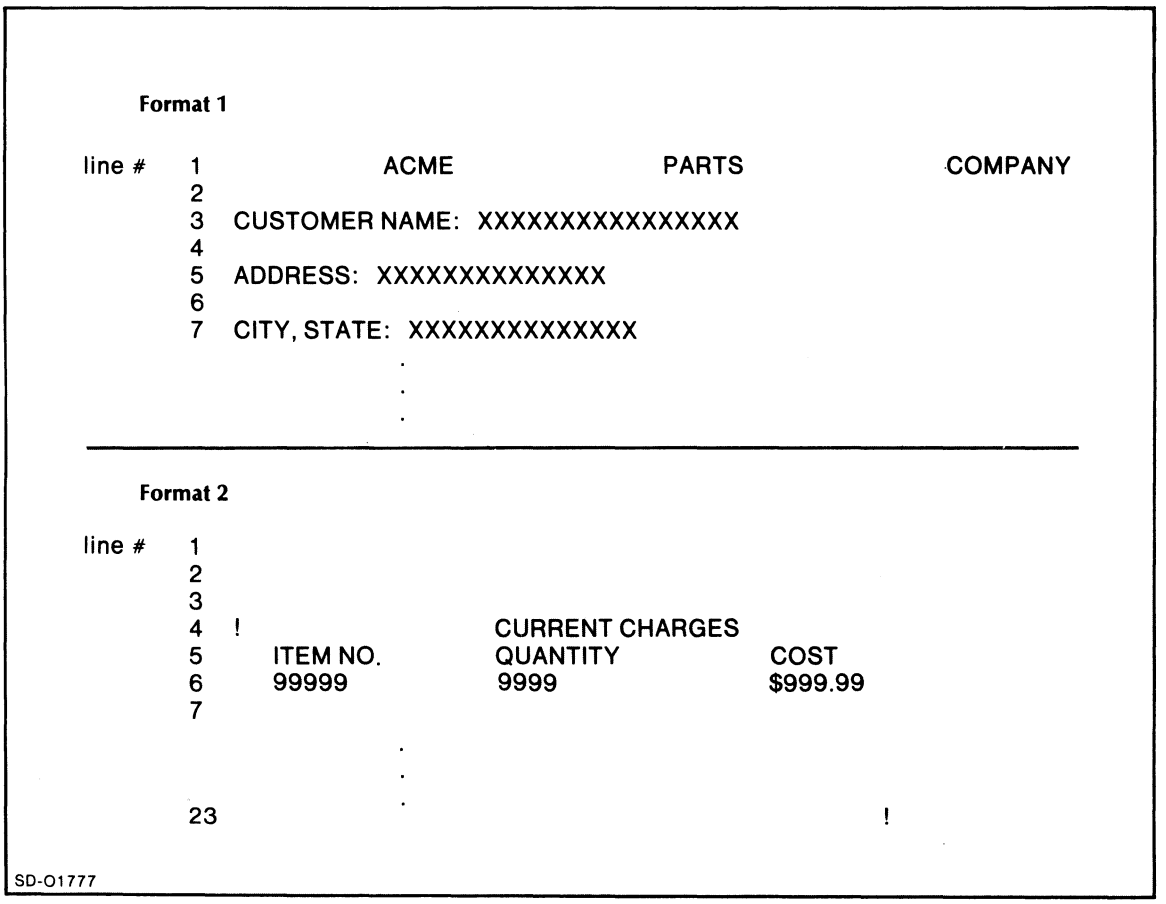


Figure 3-3. The Second Format Contains an Overlay Area

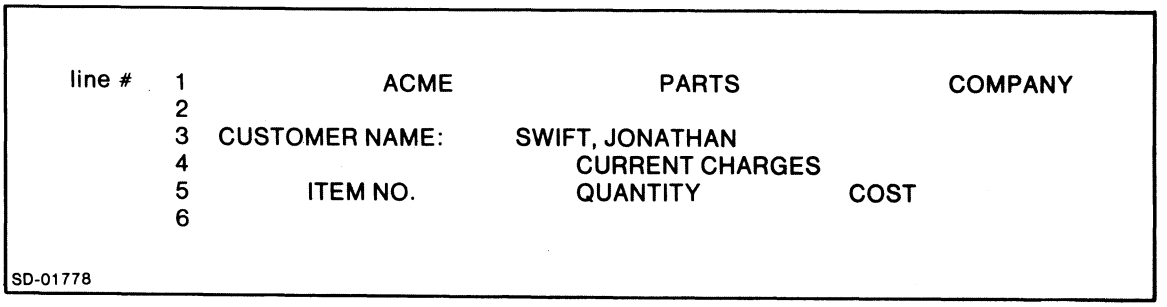


Figure 3-4. The Monitor Overlays the Area Between the Exclamation Points

Blinking Screen Text

You can cause screen literals to blink for special emphasis. While in FIELD mode, surround the literal area in square brackets.

For example, Figure 3-5 contains an area that will blink when the operator executes the format.

If you accidentally type two consecutive, identical brackets [[or]], IFMT will give the error message BRACKET USAGE INVALID.

IFMT does not check to see that each [has a corresponding], but it will automatically end the blinking at the end of the format. The blinking doesn't carry over to other screens or to messages.

You may use the left and right square brackets as literals; they control blinking only when you type them in FIELD mode.

Data fields can't blink.

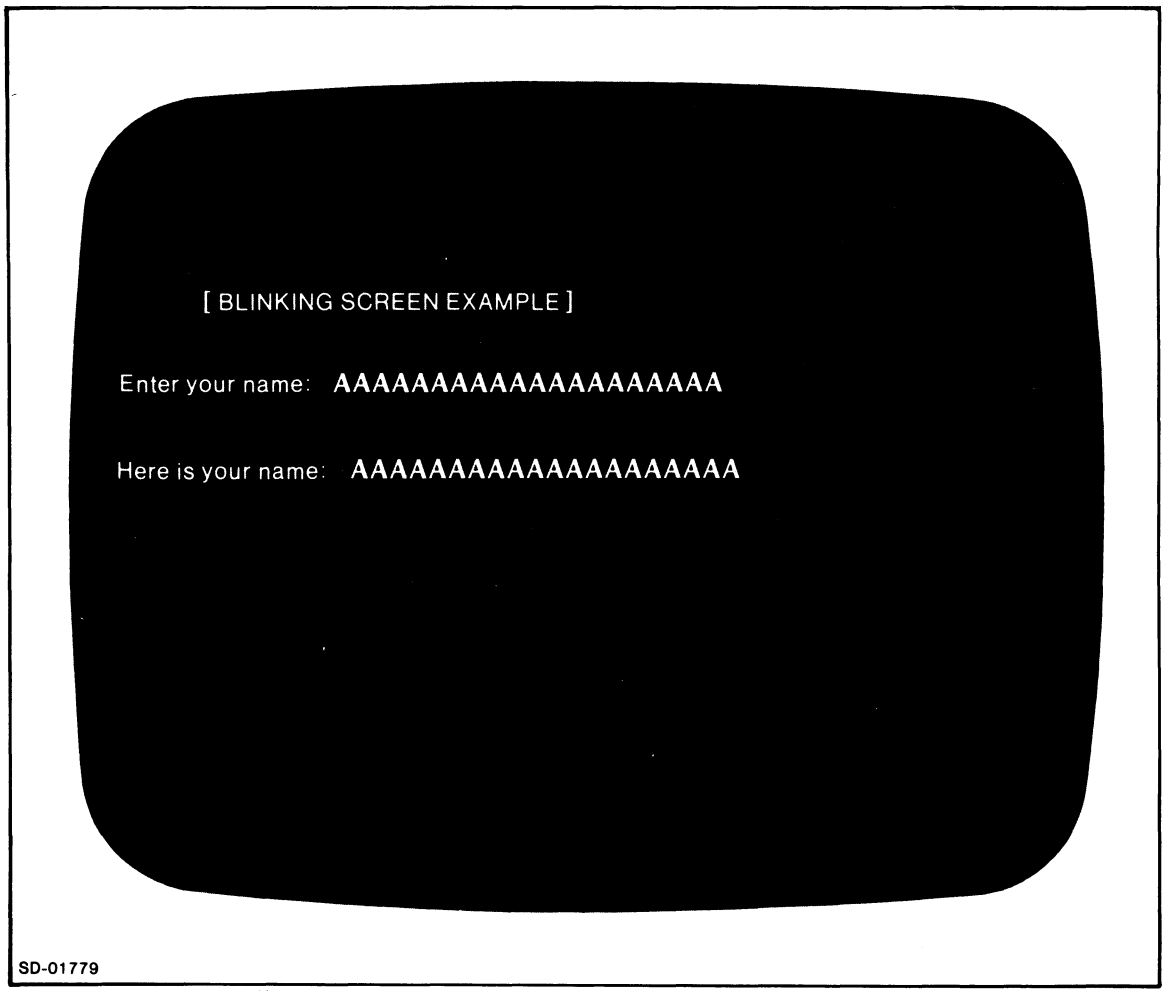


Figure 3-5. The Words BLINKING SCREEN EXAMPLE Will Blink

Underscoring Screen Information

IFMT also allows you to underscore screen literals for special emphasis. While in FIELD mode, surround the area that you want to underscore with parentheses.

For example, Figure 3-6 contains a literal that will be underscored when the operator executes the format.

If you accidently type two consecutive, identical parentheses ((or)) IFMT will give the error message BRACKET USAGE INVALID.

IFMT does not check to see that each (has a corresponding), but it will automatically end the underscoring at the end of the format. The underscoring doesn't carry over to other screens or to messages.

You can't underscore data fields.

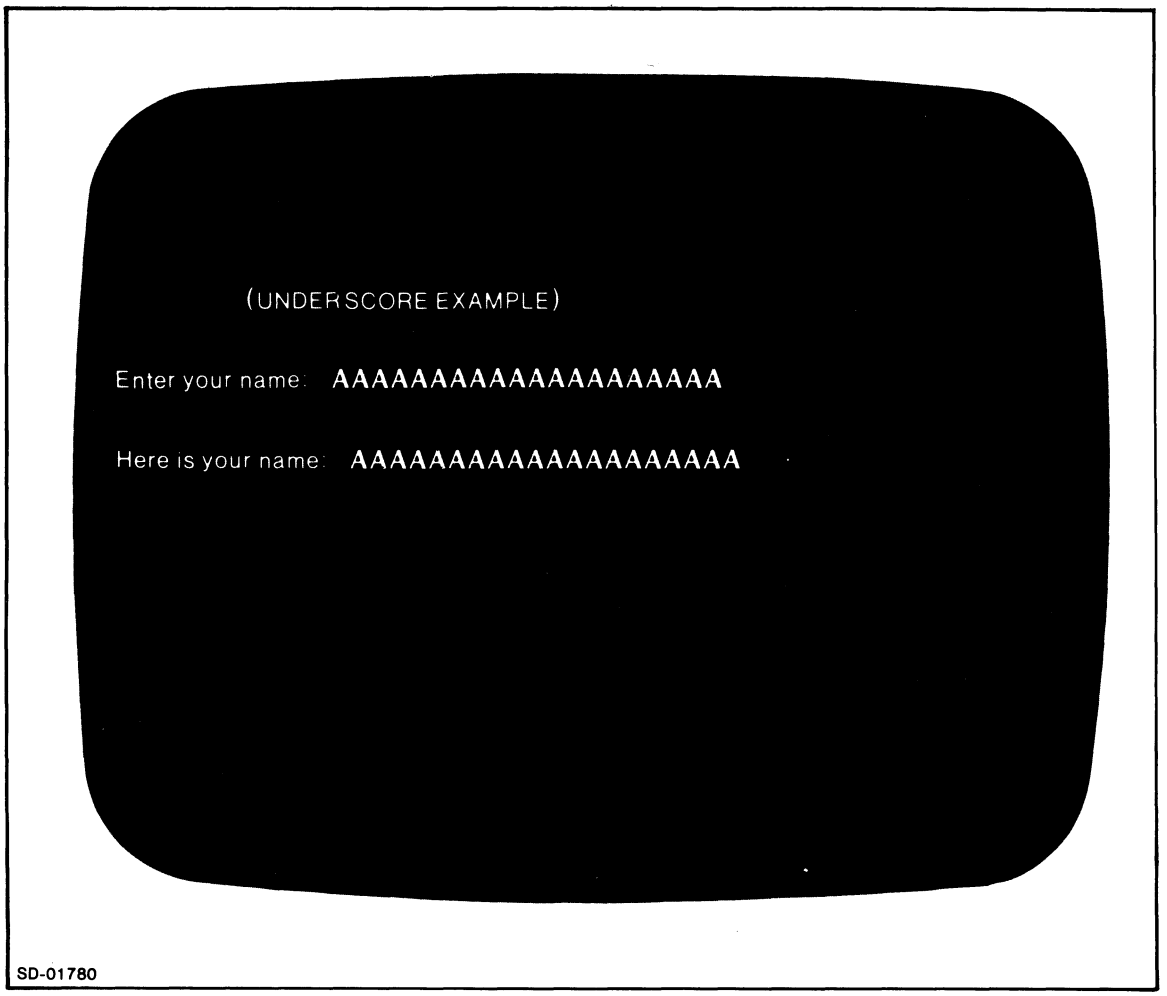


Figure 3-6. The System Underlines the Words UNDERSCORE EXAMPLE

Size and Number of Fields

A single screen format may contain a maximum of 60 data fields. Each field may be from 1 to 80 characters long, the CRT screen's maximum width.

By using SCROLL mode, you can display more than 60 fields by entering only one line to field descriptions for each set of scroll lines. The fields described on this one line are the only ones counted toward the 60 field limit. However, you will lose one field from the maximum of 60 each time you switch between page and scroll mode.

Another limitation occurs with groups. A *group* is either a scroll area or a page area, and it can contain no more than 512 characters (bytes). A scroll group will exceed the 512-byte limit if the number of lines between the @ signs multiplied by the number of field characters on one scroll line exceeds 512.

If any group exceeds 512 characters, you must divide it. To divide a scroll group in two, you insert a pair of @ signs (@@). Do not place any field specifiers between them.

Inserting the pair of @ signs is equivalent to inserting a nonfunctional page group within the scroll group. You may also use functional page groups to divide a scroll group.

To divide a page group, you can insert two successive lines containing only single @ signs. Again, this is equivalent to inserting a nonfunctional scroll group; you may also use functional groups.

Each time you divide a group, remember that it decreases the number of permissible fields by two. Also, note that you cannot backtab across a group boundary during program execution.

Attributes

After you have set up your screen literals, data fields, and scroll areas, you must assign attributes to the fields. To begin this process, strike the SHIFT and the ATTRIB key.

IFMT then checks the legality of the field definitions and the use of @ and !. If IFMT finds errors, it allows you to correct them.

If it finds no errors, IFMT displays the attribute query line at the bottom of the screen, positions the cursor to the first attribute choice, and identifies the current field by displaying flashing question marks where that field's descriptors were.

You have four possible responses to each attribute query:

Response	Meaning
Y)	You want the field to have this attribute.
N)	You do not want the field to have this attribute.
)	You want the attribute to remain as it is (the automatic default).
BACKTAB[1]	You want to return to the previous attribute for a correction.

[1] The unmarked key on the cursor key pad.

When you use BACKTAB, be sure that you are consistent with the system when choosing your attributes. For example, IFMT automatically skips the last five attribute choices if you specify a DISPLAY-only field, because DISPLAY fields can't have these attributes. However, by using BACKTAB you can change them.

On a new format, IFMT sets all attributes to N.

On a new format, IFMT sets all attributes to N.

If you are editing a previously created format, IFMT will display an attribute line with the old attributes. It will also display an asterisk after the field number in the attribute query line. You can retain these old attributes by striking NEW LINE at each one. A field's attributes will remain valid even if you change the field's size and/or data type. However, if you insert, delete, or move a field in a format, you may alter the order of processing and thus destroy the validity of the old attributes. If you are manipulating the fields in this manner, make sure that the attributes are still valid.

If you do not want to display the old attributes, you can delete the formatname.VS file with the CLI DELETE command before calling up the format with IFMT.

To verify attributes on the line printer, use the PFMT utility (see Chapter 9) or the Idea compiler (see Chapter 6).

Table 3-2 lists and describes the IFMT attributes.

Table 3-2. The IFMT Attributes

Attribute	Function
DISPLAY	The field will display data from the program. You cannot use a DISPLAY-only field for data entry; see DISPLAY and EDIT.
EDIT	The field will accept data from the operator and send it to the program.
DISPLAY and EDIT	The program will use the field as a DISPLAY field the first time it encounters it; after that, the program uses it as an EDIT field. This allows the operator to edit data from the program.
AUTO-DUP	Use this attribute for scroll fields where the fields have neither the EDIT nor DISPLAY attributes (they may have the OUTPUT attribute). An AUTO-DUP field will repeat the value that an operator first enters in subsequent scrolls of the field. CAUTION: Do not backtab to this attribute for fields with either or both the EDIT and DISPLAY attributes. If you give this attribute to fields with EDIT and/or DISPLAY, your program will not work correctly.
REQUIRED	The operator must enter at least one character in the field.
FULL	The operator must enter the exact number of characters specified by the field picture, or enter nothing.
SECURE	This attribute tells the system to echo asterisks when the operator enters characters. This ensures privacy when typing sensitive data.
AUTO-ENTRY	When full, the field supplies its own NEW LINE.

NOTE: If you designate a field as DISPLAY only, IFMT skips the last five attributes, since they do not apply to DISPLAY-only fields.

WIFMT -- The Wide Format Utility

To create print and hardcopy formats that are wider than the screen of a 6053 terminal (up to 132 characters wide), use the WIFMT utility.

To use WIFMT, give this command:

```
WIFMT)
```

WIFMT will ask you for the name of the next format, and will then ask you for the type, either print (P) or hardcopy (H).

TYPE(H OR P)

Enter H to use a DASHER printing terminal; enter P to use a line printer with PRINTF. We explain these fully in Chapter 8.

You cannot use the following IFMT capabilities with WIFMT:

- Blinks
- Underlines
- Partial Screens

How to Use WIFMT

WIFMT uses two screen lines to reach the 132-character width. It uses the 80 characters on the first line plus characters 1 to 52 on the second line. The remaining 38 characters on the second line (positions 53 to 80) are a "dead" area; WIFMT fills it with angle brackets (<).

Each two-line screen pair is a one-line unit to WIFMT; to change one line of the output format you must change both screen lines.

The two-line pairs begin at line 1, the first line of the format. Thus, odd-numbered lines mark the first 80 characters of the output format, and even-numbered lines mark the partial (52 characters) lines.

If you disturb a dead area while editing, you must repair it. Use the cursor-control keys to position to the line, and strike the BACKTAB key (the unmarked key on the cursor pad). If you are on an odd-numbered line BACKTAB will have no effect; if you are on an even-numbered line, BACKTAB will restore the dead area to its original state.

To delete a format line, you must delete both the odd- and even-numbered screen lines. Likewise, to insert a format line, you must insert a two-line pair.

WIFMT allows you to define formats that are 60 lines long (consisting of 120 screen lines). The maximum field length is 80 characters, and a field may not cross the 80th column into the 81st character position.

The PFMT utility reports format line numbers and indicates the dead area by printing a series of left angle brackets (<<<<<<<<<<<<<).

To convert IFMT formats to WIFMT, you must first insert even-numbered lines. To convert WIFMT formats to IFMT, you must delete the dead area characters; otherwise, you'll get an error message ILLEGAL CHARACTER IN FIELD. We recommend that you remake your WIFMT formats rather than convert them.

Figure 3-7 shows the screen after you give the WIFMT command, the format's name, and the H or P specification.

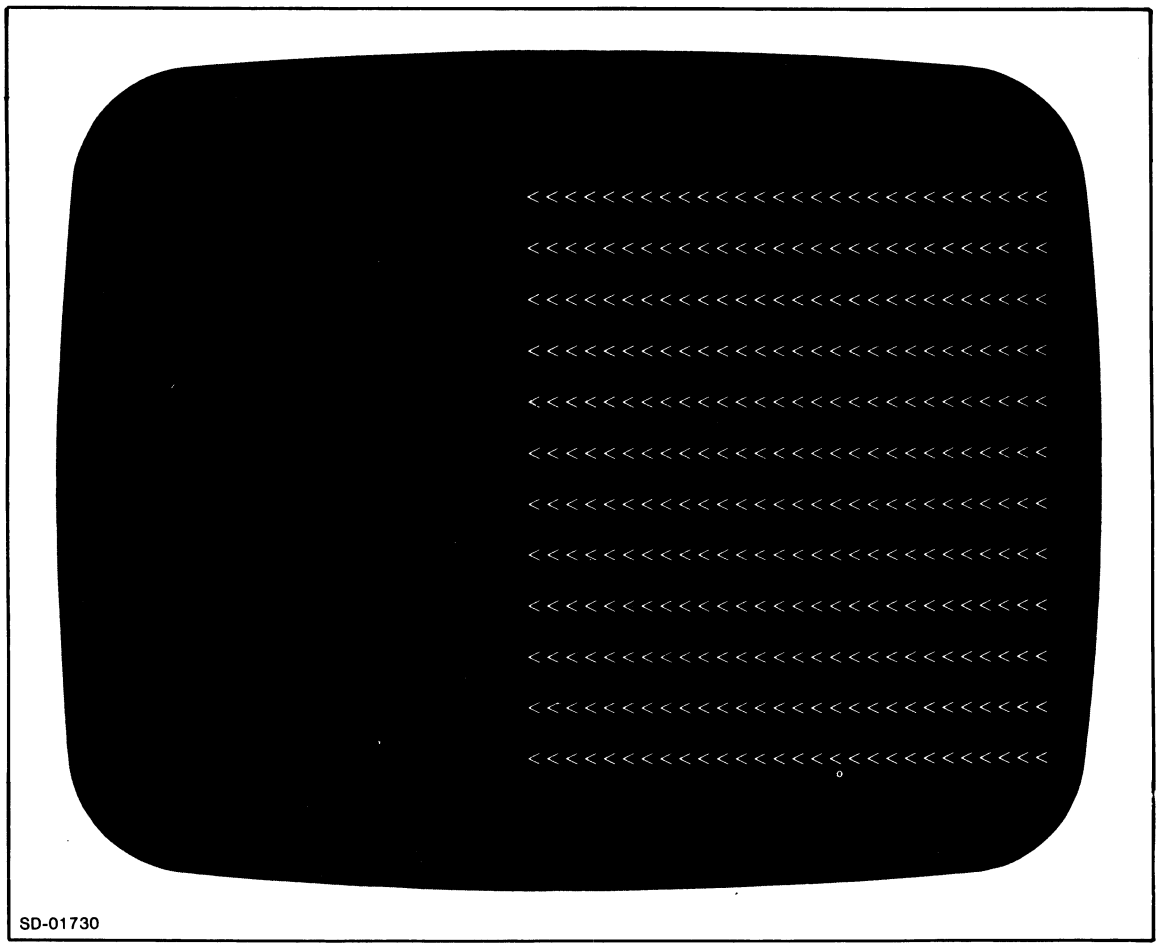


Figure 3-7. The Initial WIFMT Screen

End of Chapter

Chapter 4
The IFPL Language

Each IFPL program begins with a NAME statement and ends with a FINISH statement. Between these two statements you place groups of nonexecutable statements and groups of executable statements.

The *nonexecutable* statements perform the definition tasks for your program variable, subroutines, tables, and files. They also link the format data fields with the executable statements. Nonexecutable statements include the PROCESS statement, the REGISTER statement, the subroutine definition statements (not to be confused with routines), the table definition statements, and the file definition statements.

The *executable* statements process the variables, subroutines, tables, and files. You organize the executable statements into routines labeled by tags.

PROCESS statements direct the Idea monitor to start executing the IFPL program at these routines. The routines return control to the Idea monitor by means of RETURN, RESET, or RESTART statements.

Figure 4-1 shows the block structure of an IFPL program, and Figure 4-2 shows the structure of an actual program.

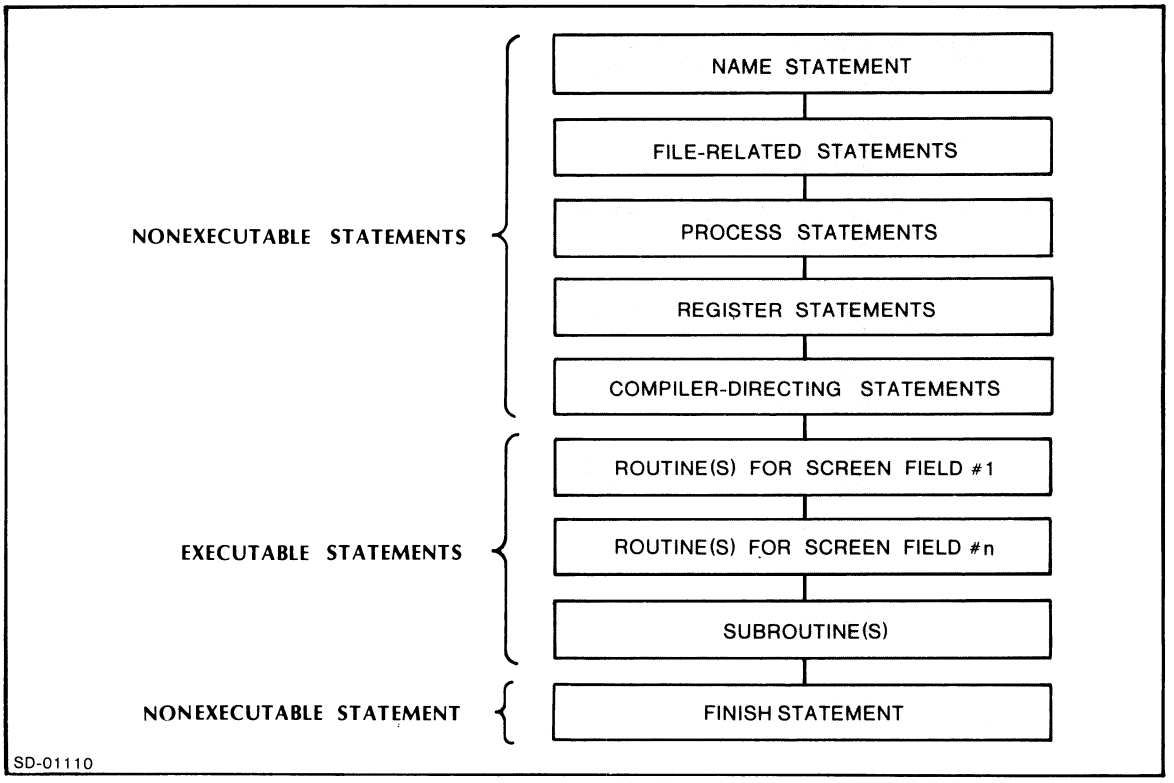


Figure 4-1. The Block Structure of an IFPL Program

The program in Figure 4-2 uses a stock item's part number (PARTNO) as a key accessing the record INVREC in the file INVENTORY. INVREC contains the item's name (PARTNAME). The program displays this name on the screen and searches the three tables (DEPTA, DEPTB, and DEPTD) for PARTNO. When the program finds PARTNO, it branches to the appropriate routine to display the department from which you can reorder the part.

```
NAME REORDER
FILE IS INVENTORY
KEY FOR INVENTORY IS 4 ASCII
RECORD FOR INVENTORY IS INVREC
        LENGTH IS 20
        INCLUDES PARTNAME 1 20 ASCII
STOP

PROCESS PARTNO AT NONE AND GETPARTNO
PROCESS PARTNAME AT DISPLAYNAME AND NONE
PROCESS DEPTNO AT DISPLDEPT AND NONE

TABLE DEPTA
"C330"
"S130"
"CS40"
ENDTABLE

TABLE DEPTB
"X250"
"Y930"
"Z280"
ENDTABLE

TABLE DEPTD
"CS30"
"M600"
"E500"
ENDTABLE

GETPARTNO:      STORE PARTNO
                RETURN

DISPLAYNAME:    FIND THE INVREC USING PARTNO
                ON-IOERR ERRMSG
                DISPLAY PARTNAME
                REFILE INVREC USING PARTNO
                ON-IOERR ERRMSG
                RETURN

DISPLDEPT:      LOOKUP IN DEPTA PARTNO
                IF FOUND D1
                LOOKUP IN DEPTB PARTNO
                IF FOUND D2
                LOOKUP IN DEPTD PARTNO
                IF FOUND D3

D1:             MOVE "DEPT A" TO DEPTNO
                DISPLAY DEPTNO
                RETURN

D2:             MOVE "DEPT B" TO DEPTNO
                DISPLAY DEPTNO
                RETURN

D3:             MOVE "DEPT D" TO DEPTNO
                DISPLAY DEPTNO
                RETURN

ERRMSG:         MESSAGE IO-ERROR.  CALL SYSTEM MANAGER.
                QUIT

FINISH
```

Figure 4-2. An IFPL Program

Nonexecutable Statements

The nonexecutable statements include the PROCESS statement, the REGISTER statement, the subroutine definition statements[1], the table definition statements, and the file defintion statements.

The PROCESS Statement

The PROCESS statement controls the execution sequence of an IFPL program. You must have one PROCESS statement for each DISPLAY or EDIT field. Also, you must place your PROCESS statements together in a group with no intervening statements.

The compiler links the first DISPLAY or EDIT field it encounters to the first PROCESS statement, the second DISPLAY or EDIT field to the second PROCESS statement, and so forth. The order of the fields on the screen runs from left to right and from top to bottom. If you mix up this order, you'll get meaningless program results, since the compiler will link the fields to the wrong routines.

The formal syntax of the PROCESS statement is

[label#] PROCESS variable AT { tag *[AND]* NONE | NONE *[AND]* tag }

The optional *label* lets you send program control, via a RETURN statement or some other statement, to a section of code identified by a PROCESS statement. If you use a label, you must place a # sign immediately after the label, with no spaces in between the label and the # sign. Labels can be up to 10 characters long.

variable is the name you want to give to your program variable; it must be unique within the program. The compiler assigns this variable an area of working storage with characteristics defined by the field attributes. (At runtime, though, the variable is not connected to the field.)

For a DISPLAY-only field, use AT tag AND NONE; **tag** is the label of the routine to which you want this PROCESS statement to pass execution. For example, the statement

```
PROCESS PARTNAME AT DISPLAYNAME AND NONE
```

sends program execution to the routine labeled DISPLAYNAME.

For an EDIT-only field, use AT NONE AND tag. The PROCESS statement

```
PROCESS PARTNO AT NONE AND GETPARTNO
```

sends execution to the routine labeled GETPARTNO.

For a field with both the DISPLAY and EDIT attributes, use AT tag1 AND tag2. **tag1** is the label of a routine that will use the field as a DISPLAY field, and **tag2** is the label of a routine that will use the field as an EDIT field. The first time the system encounters the field, it uses the field as a DISPLAY field; after that, it uses the field as an EDIT field (unless you change this with a RESET statement).

---

[1] Strictly speaking, the subroutine definition statements are executable statements, but they must not appear in an executable block, such as a routine.

The REGISTER Statement

The REGISTER statement is another way to declare a program variable. A variable declared by a REGISTER statement is identical to one defined by a PROCESS statement. You will use REGISTER-statement declared variables to define temporary storage that is independent of the screen.

The formal syntax of the REGISTER statement is

REGISTER variable picture *[initial value]*

where:

variable — is the name you want to assign to your program variable. It must be unique within the program.

picture — is a picture of your variable. If you will use this variable with a screen field for storing or displaying data, this picture must correspond to the picture of the variable that appears in the screen format.

initial value — is optional. If you assign an initial value, it must correspond to the variable's picture; i.e., you can't assign an initial numeric value if your variable picture is alphabetic.

Subroutine Definition Statements

A subroutine is a group of executable statements that is not connected to a screen field with a PROCESS statement. To execute a subroutine, give the PERFORM statement.

To define the beginning of a subroutine, use this statement:

SUBROUTINE subroutinename

To end a subroutine, use the statement

ENDSUB

This statement also returns execution to the main program.

Table Definition Statements

Use this statement to define the beginning of a table:

TABLE tablename

To end a table, use the statement

ENDTABLE

File Definition Statements

To use a file in an IFPL program,

1. You must create the file using the INFOS utility ICREATE (see Chapter 5), and

2. You include certain file definition statements. These file definition statements are:

```
FILE [IS] filename
KEY [FOR] filename [IS] length ASCII
RECORD [FOR] filename [IS] recordname
LENGTH [IS] recordlength
INCLUDES fieldname starting-position length type

STOP

DUPLICATES [ARE] COUNTED [IN] variable
PARAMETERS [FOR] subindexname
NODE-SIZE [IS] value
PARTIAL LENGTH [IS] VALUE

SUBINDEX [FOR] filename [IS] subindexname

DEFINE subindexname USING key...
```

Executable Statements

You group the executable statements in routines that you connect to the screen data fields via tags in the PROCESS statements. Label the first statement in the routine with the same tag that you used in the PROCESS statement, ending the tag with a colon (:).

Terminate the routine with a LINK, RETURN, RESET, or RESTART statement. The LINK statement links to another format. RETURN goes to the next PROCESS statement; if the routine has completed the last PROCESS statement, control passes to the FINISH statement. RESTART returns control to the first field, erases all unprotected data, and resets the DISPLAY/EDIT flip-flop to DISPLAY (for fields with both DISPLAY and EDIT). RESET resets a field with both DISPLAY and EDIT to DISPLAY.

As with other IFPL names, tags must begin with a letter. The remaining characters can be any combination of letters, numbers, dashes (-), and periods (.) (See the section on "Names" in this chapter).

You cannot place a space between the tag and the colon.

E1: is a legal tag; E2 □ : is illegal.

The set of executable statements allows you to perform arithmetic functions, control functions, data moves between the screen and the program, data manipulation, file manipulation, passing, sending/receiving, and printing. For each statement's formal syntax, see Chapter 7.

Data Moves Between Screen and Program

The STORE statement takes a value entered in a screen EDIT field and stores it in working storage. You must give this command to use data entered on the screen in your program.

The DISPLAY statement displays the value of a program variable on the screen in a field that has the DISPLAY attribute.

Arithmetic Functions

The arithmetic function statements are ADD, SUBTRACT, MULTIPLY, and DIVIDE. They all take this form:

operator **value**$_1$ **value**$_2$ **resultvariable**

The operator performs the arithmetic function using **value**$_1$ and **value**$_2$, and places the result in **resultvariable**. The SUBTRACT statement subtracts **value**$_1$ from **value**$_2$; the DIVIDE statement divides **value**$_1$ by **value**$_2$.

Be conscious of possible truncation problems when you define your **resultvariable**. The monitor will round off decimal fractions and will truncate the left digits of integer values if you haven't provided enough digits to the left of the decimal point. If such an overflow occurs, the monitor sets the overflow flag. You can use this flag with the ON-OVERFLOW statement to branch to an error-handling routine (see ON-OVERFLOW in Chapter 7).

For example, suppose you declared your **resultvariable** with this picture:

```
99.99
```

This addition,

```
 10.1111
+ 1.1171
 11.2281
```

would become 11.23. And this addition,

```
100.1111
+ 1.1171
101.2282
```

would become 01.23.

To ensure that you don't lose valuable digits, follow these rules:

ADD	Give the **resultvariable** one more integer place than the larger addend.
SUBTRACT	Give the **resultvariable** one more integer place than the larger of the minuend and subtrahend.
MULTIPLY	Give the **resultvariable** as many integer places as the multiplier plus the multiplicand.
DIVIDE	Give the **resultvariable** as many integer places as the dividend and as many decimal places as the divisor.

Internal Considerations

The monitor will perform arithmetic on up to 18 decimal places. It performs all calculations in real arithmetic and assumes a decimal point after the right-most digit if you don't specify one. The decimal point is implicit in all cases; you don't have to provide a character position for it.

Signed Values

You must define your **resultvariable** as a signed variable, or the sign will be lost. The sign requires one character position.

Control Statements

The control statements are RANGE, COMPARE, LOOKUP, GO TO, and ON-IOERR.

The RANGE statement checks to see if a value is within a certain range. If it is, you can use the IF IN-RANGE statement to direct program execution; if it isn't, you use the IF OUT-RANGE statement.

The COMPARE statement compares two values and sets a flag according to what it finds. The IF EQUAL, IF NOT-EQUAL, IF LESS, and IF GREATER statements direct program execution according to the value of the flag.

The LOOKUP statement searches a table for a value and sets a flag. The IF FOUND and IF NOT-FOUND statements direct program execution according to the flag's value.

The GO TO statement is an unconditinal GO TO; the GO TO USING statement is a conditional GO TO.

To branch to an I/O error-handling routine, you use the ON-IOERR statement.

You can use 11 other control statements to handle special conditions that may arise during data entry. For example, the ON BACKTAB statement branches to a routine if the operator strikes the BACKTAB key (the unmarked key on the cursor pad).

These additional statements are:

ON BACKTAB
ON DISCONNECT
ON END DATA
ON ESCAPE
ON FUNCTION
ON LINE-ERR
ON LOGOFF
ON MODE CHANGE
ON NO-ACTIVITY
ON REPEAT
ON SCREEN

Data Manipulation Statements

You can transfer data between memory locations with the MOVE statement, the RIGHT statement, and the LEFT statement. If you use these statements with tables, be sure that the source and destination tables have the same data types and sizes. See Chapter 7 for more information on these statements.

File Manipulation Statements

To locate a file record and bring it into memory, use a variation of the FIND statement.

To enter a new record into the database, use the FILE-NEW statement; to update a record, use the REFILE statement.

To delete a record permanently, use the DESTROY statement. To delete a record logically, use the REMOVE statement. To recover a logically deleted record, use the REINSTATE statement.

To lock a record, use the HOLD keyword in the FIND statement. To release a locked record for use by other programs, use the RELEASE statement. The RELEASE statement also allows you to unlock all records locked by the program.

To verify that a key will retrieve a record you can use the VERIFY statement. The system will set the IOERR flag as if a record access was attempted. VERIFY does not, however, retrieve the record. This is useful for positioning within INFOS system sublevels. You can also use VERIFY NEXT or VERIFY PREVIOUS to look beyond a record that was locked by another program.

The RETRIEVE key and RETRIEVE HIGH key statements let you place key values in variables.

The ESTABLISH LINK statement sets up a link between a key and a subindex.

The INVERT statement lets you set up an alternate key for a record.

Printing Statements

The printing statements are:

RECORD *[FOR]* PRINTING *[IS]* recordname

INITIATE PRINTING USING key

PRINT *[THE]* recordname USING key

TERMINATE PRINTING USING key

We explain these fully in Chapter 8.

Sending and Receiving Data

To send data to another process, use a form of the SEND statement:

SEND { recordname *[[TO] ipc-portname]* / REQUEST recordname }

To receive data sent from another process, use:

RECEIVE recordname *[[FROM] ipc-portname]*

You may use RCX70 to send and receive data. If you elect this option (see IDEASG in Chapter 10), Idea will perform several tasks to make such communication simple. The system will set up IPC headers, split records if they exceed the RCX70 buffer size, and attach a valid RCX70 command code and address. You must simply place in the record the information that the host expects or returns.

Statements for Tape Logging

Use these statements for tape logging:

RECORD *[FOR]* TAPE *[IS]* recordname

LOG *[THE]* recordname

Passing Records to Another Program

To pass a record to the COMMON area so that a linked program can accept it, use:

RECORD *[FOR]* PASSING *[IS]* recordname

PASS recordname

To accept a record from the COMMON area, use

RECORD *[FOR]* PASSING *[IS]* recordname

ACCEPT recordname

Miscellaneous Statements

The statement

COPY filename

copies the contents of a file into a program.

The statement

PRIORITY *[IS]* value

assigns a lower processing priority to the program.

The statement

QUEUE variable

queues a CLI command as a batch job.

Names

You must follow certain conventions when assigning names to your programs, variables, tables, files, records, and tags.

Program Names

Program names in the NAME statement must begin with a letter; the remaining characters may be letters, numbers, or periods (.). You may not use the following characters in the NAME statement:

dash	-
colon	:
carat	^
single quote	'
double quote	''
angle brackets	< >
parentheses	( )

Program names can contain any number of characters; however, the first 10 must be a unique name.

Other Names

Names for variables, tables, files, records, and tags must begin with a letter. The remaining characters can be letters, numbers, periods, dashes, or other punctuation characters except the following:

colon	:
carat	^
double quote	''

The colon serves as the tag delimiter. You must place a colon immediately after a tag, and follow the colon with at least one space or tab. For example,

```
MYTAG:  STORE NAME
        RETURN
```

The carat is the line continuation character, and the double quote encloses literals, such as "A_LITERAL".

Length

You can specify any number of characters in your names, but the first 10 must be unique.

Delimiters

To separate a name from a keyword or another name, use a space, a tab, or a comma.

Statements That Define Names

The following statements define names. The name in each statement is underlined.

NAME programname

FILES filename...

REGISTER variable picture *[initial value]*

RECORD FOR { filename / PASSING / PRINTING / TAPE } IS recordname

TABLE tablename

SUBROUTINE subroutinename

PROCESS variable AT { tag1 AND NONE / NONE AND tag2 }

Using the REDESIGNATE Statement

You can use the REDESIGNATE statement to define a name. A register redesignation is equivalent to a register declaration; the names specified in the redesignation define valid names. For example,

```
REGISTER DATE X (8) 00/00/00
REDESIGNATE DATE
MONTH 1 2
DAY 4 2
YEAR 7 2
STOP
```

In this example, we redesignated a portion of the register DATE as MONTH, a portion as DAY, and a portion as YEAR. Another possible way to define this register would be

```
REGISTER MONTH X (2) 00
REGISTER DAY X(2) 00
REGISTER YEAR X(2) 00
STOP
```

However, the values of MONTH, DAY, and YEAR receive six contiguous bytes of storage when you REDESIGNATE them. If you declare them as three separate registers (as in the second case), they aren't stored contiguously.

Data Types

IFPL has three data types: numeric, alphabetic, and alphanumeric. They are defined by their character sets:

Numeric: 0-9 . + -

Alphabetic: The characters contained in the file ALPHABET.TB. For English-speaking users, this set will usually consist of the letters A-Z and the space.

To change the alphabet, use the ALPHA utility described in Chapter 9.

Alphanumeric: Any keyboard character.

IFPL allows you to use these data types in registers. It obtains the data for PROCESS variables from the format field definitions.

Auxiliary Words

You may use the following words in statement lines or you may leave them out. In the statement descriptions in Chapter 7, we enclose these optional words in square brackets:

AND	IF	OF
ARE	IN	THE
AT	IS	TO
FOR	JUSTIFY	WITH
FROM	ON	

You cannot define any of these words as a name.

Continuation Lines

To continue a statement onto another line, end the first line with a ^ character (keys SHIFT and 6). Begin the second line flush left -- the compiler will see a space or a tab as a break between two words. You may use only one continuation line per statement.

Example

```
MESSAGE CUSTOMER NO. ALREADY ON FILE,EN^

        TER 'R' TO ACCESS RECORD.
```

(This is incorrect.)

```
MESSAGE CUSTOMER NO. ALREADY ON FILE,EN^

TER 'R' TO ACCESS RECORD.
```

(This is the correct form.)

Comments

To place a comment in an IFPL program, begin the comment with an asterisk (*). The compiler recognizes information following the asterisk as a comment and will not try to interpret it.

You can begin a comment anywhere on a line. However, you may not place a comment in a MESSAGE statement, since the asterisk is interpreted as being part of the message.

Also, you may not use a comment in a REGISTER statement that defines an alphanumeric or alphabetic register; the system interprets the asterisk as part of the register's initial value.

You may use a comment in a REGISTER statement that defines a numeric register; the asterisk terminates the numeric portion of the register and begins the comment.

Sending Control Characters

You can send control characters directly to a terminal from an IFPL program without filtering or interpretation by the Idea terminal interface routines. To do this, enclose the angle-bracketed control characters in exclamation points (!). The first exclamation point disables interpretation, the second one re-establishes it.

You can send control characters via the MESSAGE statement, or you can use a nonnumeric REGISTER statement to set up a register:

```
MESSAGE !<47><57>! THIS IS A MESSAGE

REGISTER REGA X (9) ABCDER!<47>!
```

Each exclamation point occupies one byte of storage. If the output of control characters results in the loss of the terminal's cursor position, use one of the following sequences to correct the problem:

1. Before turning interpretation back on, send positioning codes to restore the cursor position.

2. Turn the interpreation back on and send the control sequence <375> <320> <row> <col>.

Note that with the second method, you do not use the exclamation point delimiters. The codes are not actually sent to the terminal, but are intercepted by the monitor. The initial code <375> signals this interception. The monitor then reads the codes and positions the cursor.

Reserved Words

Table 4-1 lists the special registers that you may use in IFPL programs. The monitor initializes these registers every time it enters a program to process a screen field. You must declare all but three of these registers in the program. You may use them just as you would any register.

Table 4-1. IFPL Reserved Words and Their Pictures

BATCH	X(3)	FUNCTION	9(1)	MONTH	9(2)
CHARACTERS	9(2)	HOURS	9(2)	PASSWORD	X(10)
CRT	9(2)	INFOS-ERR[1]	9(3)	SECONDS	9(2)
DAY	9(2)	IOERR[1]	9(2)	VARIED-KEY[2]	
ENTRY[1]	9(2)	MINS	9(2)	YEAR	9(2)
FIELD	9(2)				

[1] Defined by the compiler. Define all others in REGISTER or PROCESS statements.

[2] Takes any picture you specify.

Reserved Word	Explanation
BATCH	This register contains the batch number associated with disk and tape logging systems. The operator supplies this number in response to a system question when logging on. If you define the BATCH register in the program, it gets this operator-defined value.
CHARACTER	This register contains the number of characters entered in the last EDIT field. However, it doesn't count NEW LINE as a character. The system updates this value after every EDIT field.
CRT	This two-byte word returns the AOS console number of the CRT.
PASSWORD	This 10-byte word returns the password most recently used to log on the terminal running the program.
MONTH, DAY, YEAR	These two-byte words contain the system month, date, and year.
HOURS, MINS, SECONDS	These words return the system hours, minutes, and seconds. They are updated at each entry to the program.
FIELD	This word contains the physical number of the current field.
ENTRY	The system sets this two-byte register to the index value of a table element according to the findings of a LOOKUP statement. You can then use ENTRY in a control statement.

Table 4-1. IFPL Reserved Words and Their Pictures (continued)

Reserved Word	Explanation
INFOS-ERR	Following a database access or a SEND/RECEIVE, this register contains the actual INFOS or AOS error code. The system updates this register after such statement.
IOERR	Whenever the program attempts to access a data file, this register receives one of the error codes listed below.
FUNCTION	When the operator strikes one of the user-defined function keys, the system places a number in this register. The numbers are 1, 2, 3, or 4, corresponding to the number of the function key. (The numbers run 1-2-3-4 from left to right.)
VARIED-KEY	Use this register with RETRIEVE KEY and RETRIEVE HIGH KEY statements to accept the value of the key. Give VARIED-KEY as many characters as the largest key you will store in it. The system will delete spaces in VARIED-KEY so that it matches the exact length of the key you retrieve. If you don't specify enough spaces, the system will truncate the value to fit the register.

End of Chapter

Chapter 5
Using INFOS Files with Idea Programs

Idea programs use INFOS® system DBAM files. Before you read this chapter you should read the *INFOS System User's Manual (AOS),* 093-000152, which explains the various options available with INFOS system files, such as duplicate keys, generic keys, approximate keys, inverted keys, and subindexes. It also shows you the best file structure to use for each type of application.

In this chapter, we will demonstrate how to create a single-key DBAM file with the ICREATE utility. (You must use this utility to create files; you cannot create a file from within a program.) We will then use the file with two programs: one to load the database and one to update it. These programs will demonstrate IFPL's file definition and file manipulation statements.

Creating a File

We will create the simplest type of INFOS system file used with Idea -- a single-key DBAM file. The file will consist of records containing two fields: part name and initial quantity. We will use each part's number as the key. (The key does not have to be a field within the record, as we will demonstrate.) Each part will have only one part number, so each key will uniquely identify one part record.

Figure 5-1 shows our file with the key values included as fields within the record. Figure 5-2 shows the file we will create; the keys are not included within the records.

Notice that an INFOS file consists of two files: a database file containing the records and an index file containing the key values.

Our file's name will be INVENTORY. We will not allow duplicate keys in the index, since each part number uniquely identifies one part. Also, we will not use partial records or any of the other INFOS system options.

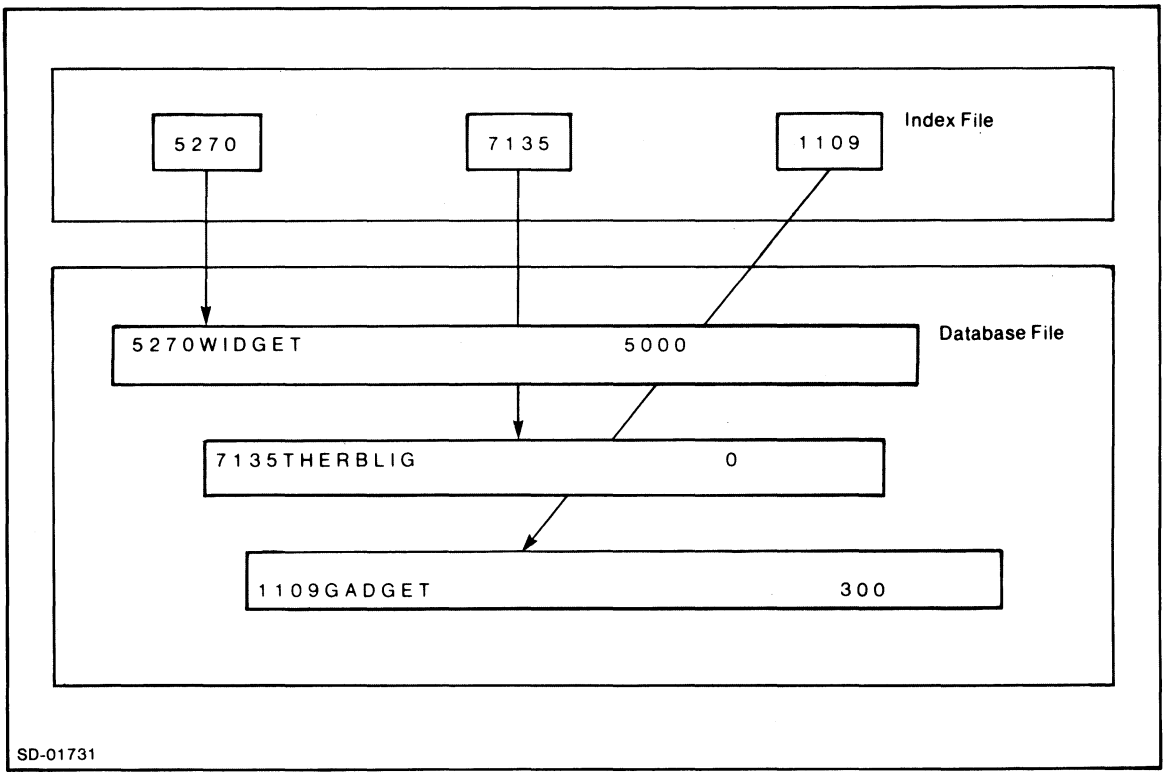


Figure 5-1. A Single-Key ISAM File Where the Key IS a Field in the Record

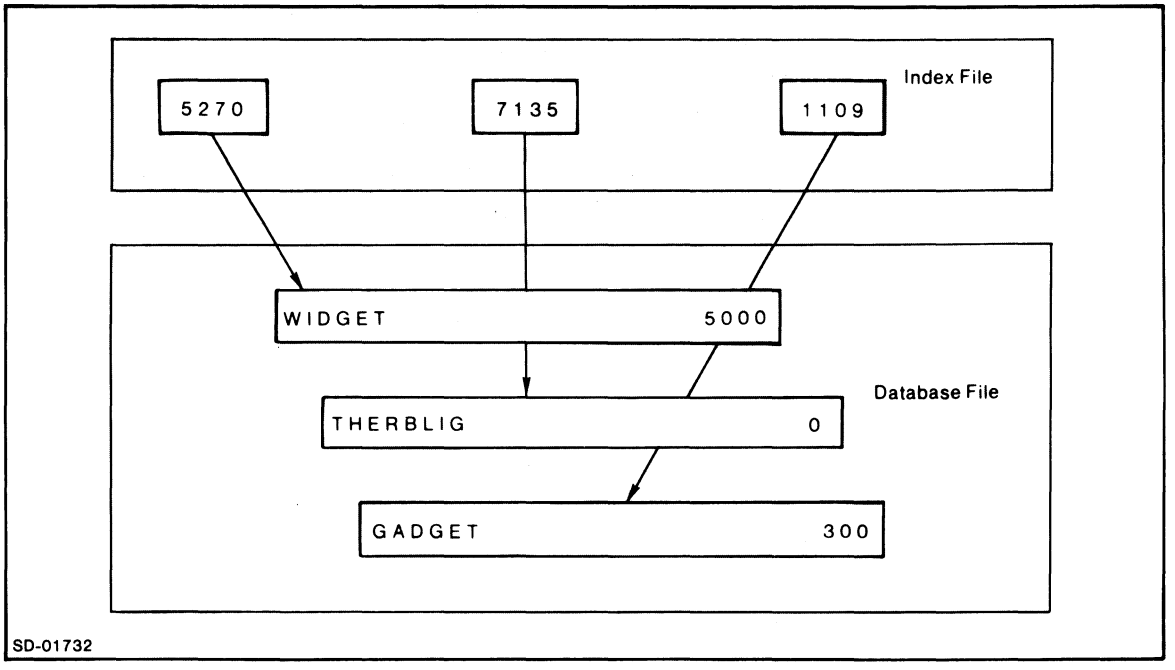


Figure 5-2. A Single-Key ISAM File Where the Key Is NOT Part of the Record

To create the file, give this command from the CLI:

)ICREATE)

This begins a dialog with the system, shown in Figure 5-3.

```
NAME OF FILE TO BE CREATED: INVENTORY)
ACCESS METHOD (I=ISAM, D=DBAM) [D]:)

 ****** DEFINE INDEX FILE ******

        MAXIMUM NUMBER OF INDEX LEVELS [2]:)
        PAGE SIZE (BYTES) [2048]:)
        PARTIAL RECORD LENGTH [0]:)
        ROOT NODE SIZE [2042]:)
        MAXIMUM KEY LENGTH [255]:)
        ALLOW DUPLICATE KEYS IN THIS INDEX? (Y OR [N]):)
        ENABLE SPACE MANAGEMENT? (Y OR [N]):)
        ENABLE KEY COMPRESSION (Y OR [N]):)
        OPTIMIZE RECORD DISTRIBUTION (Y OR [N]):)

         ****** DEFINE INDEX VOLUME(S) ******

        NUMBER OF VOLUMES TO DEFINE [1]:)
        VOLUME 1 NAME [VOL01]:)
                SPECIFY MAXIMUM SIZE? (Y OR [N]):)
                SPECIFY FILE ELEMENT SIZE? (Y OR [N]):)

 ****** DEFINE DATABASE FILE ******

        DATABASE FILE NAME [INVENTORY.DB]:)
        PAGE SIZE (BYTES) [2048]:)
        ENABLE SPACE MANAGEMENT? (Y OR [N]):)
        ENABLE DATA RECORD COMPRESSION (Y OR [N]):)
        OPTIMIZE RECORD DISTRIBUTION (Y OR [N]):)

         ****** DEFINE DATABASE VOLUME(S) ******

        NUMBER OF VOLUMES TO DEFINE [1]:)
        VOLUME 1 NAME [VOL01]:)
                SPECIFY MAXIMUM SIZE? (Y OR [N]):)
                SPECIFY FILE ELEMENT SIZE? (Y OR [N]):)
```

Figure 5-3. Our Dialog with ICREATE

Creating a Program to Build the Database

After creating the file, we will write a program to build the database. We want to enter three values into the program -- PARTNO, PARTNAME, and QUANTITY -- and use PARTNO as the key to store PARTNAME and QUANTITY as fields in the record. We also want to restart the program by entering a Y in response to a screen literal prompt, DO YOU WANT TO ENTER ANOTHER PART? (TYPE Y OR N, THEN NEW LINE). This requires an extra field.

Figure 5-4 shows the screen named NEWPART. We will give all four fields the EDIT attribute since we will enter values into the program through them.

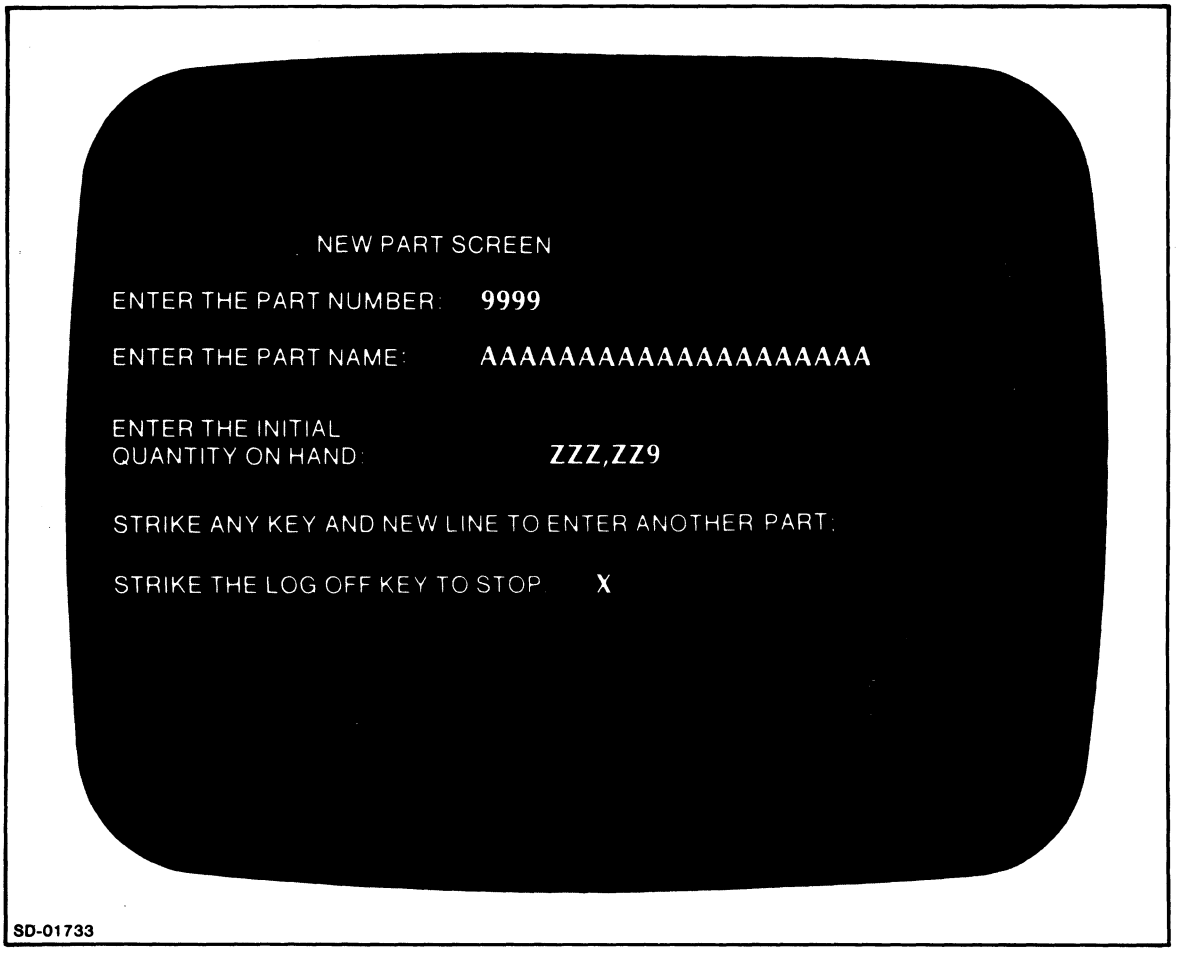


Figure 5-4. The Screen Format Named NEWPART

Figure 5-5 shows the program that will build the database.

File Definition Statements in NEWPART.UP

In Figure 5-5, the statements from FILE IS INVENTORY to STOP define our file, key, and record. Each IFPL program that uses a file must contain a block of statements similar to the one in our example.

FILE IS INVENTORY

Gives the name of the file. We use the name that we gave to the file when we created it with ICREATE.

KEY FOR INVENTORY IS 4 ASCII

Specifies that the key is 4 characters long.

```
NAME NEWPART.UP

FILE IS INVENTORY
KEY FOR INVENTORY IS 4 ASCII

RECORD FOR INVENTORY IS QONHAND
        LENGTH IS 26
        INCLUDES PARTNAME 1 20 ASCII
        INCLUDES QUANTITY 21 6 ASCII
STOP

PROCESS PARTNO AT NONE AND GETPARTNO
PROCESS PARTNAME AT NONE AND GETPARTNAME
PROCESS QUANTITY AT NONE AND GETQUANT
PROCESS NEWSCREEN AT NONE AND NEXTPART

GETPARTNO:      STORE PARTNO
                RETURN

GETPARTNAME:    STORE PARTNAME
                RETURN

GETQUANT:       STORE QUANTITY
                FILE=NEW QONHAND USING PARTNO
                ON=IOERR ERRMSG
                RETURN

NEXTPART:       STORE NEWSCREEN
                RETURN

ERRMSG:         MESSAGE I/O ERROR.  CALL SYSTEM MANAGER.
                QUIT

FINISH
```

Figure 5-5. The Program NEWPART

RECORD FOR INVENTORY IS QONHAND

Begins the record definition block for the record named QONHAND.

LENGTH IS 26

Gives the overall length of the record QONHAND in bytes (characters).

INCLUDES PARTNAME 1 20 ASCII

Defines the first field in the record, which is named PARTNAME. The number 1 says to begin PARTNAME at the first character position in the record QONHAND. The number 20 is PARTNAME's length in bytes. The keyword ASCII indicates that the information is in regular ASCII character format. This is the most common format; for other options, see the INCLUDES statement description in Chapter 7.

INCLUDES QUANTITY 21 6 ASCII

Defines the second field in the record, which is named QUANTITY. The number 21 is the starting position of this field; the number 6 is its length in bytes.

STOP

Ends the record definition block for the record QONHAND.

File Manipulation Statements in NEWPART.UP

In a database-loading program, there is only one file manipulation statement -- the FILE-NEW statement. In this program, the statement

```
FILE-NEW QONHAND USING PARTNO
```

creates a new QONHAND record that you will access later by using the value now in the variable PARTNO as the key.

Notice the ON-IOERR ERRMSG statement with the ERRMSG routine shown previously in Figure 5-5. You should place an ON-IOERR statement immediately after each file manipulation statement in a program to check for errors.

Creating a Program to Update the Database

After we create our database-building program and run it to create our records, we will need another program to access the database and update it. In our example, we will create a program that will update the QUANTITY field whenever production releases a batch of parts to inventory.

Figure 5-6 shows the screen named QUPDATE. This screen, along with the program in Figure 5-7, will take a part number that we enter, find the corresponding part record, and display the part name as a check to ensure that we are updating the correct record. Then, we input the quantity of the part that has arrived from production. The program adds this quantity to the old quantity, updates the record, and displays the new quantity on hand.

Figure 5-6. The Screen Format Named QUPDATE

File Definition Statements in QUPDATE.UP

In Figure 5-7, the file definition statements in QUPDATE.UP are identical to those in NEWPART.UP. They didn't have to be; if we hadn't used the PARTNAME field in QUPDATE.UP, we could have omitted the statement INCLUDES PARTNAME 1.20 ASCII and simply specified INCLUDES QUANTITY 21 6 ASCII.

```
NAME QUPDATE.UP

FILE IS INVENTORY
KEY FOR INVENTORY IS 4 ASCII

RECORD FOR INVENTORY IS QONHAND
        LENGTH IS 26
        INCLUDES PARTNAME 1 20 ASCII
        INCLUDES QUANTITY 21 6 ASCII
STOP

PROCESS PARTNO AT NONE AND GETPARTNO
PROCESS PARTNAME AT DISPLAYNAME AND NONE
PROCESS NEWQUANT AT NONE AND GETQUANT
PROCESS QUANTITY AT DISPQUANT AND NONE
PROCESS NEWSCREEN AT NONE AND NEXTPART

GETPARTNO:      STORE PARTNO
                RETURN

DISPLAYNAME:    FIND AND HOLD THE QONHAND USING PARTNO
                ON=IOERR ERRMSG
                DISPLAY PARTNAME
                RETURN

GETQUANT:       STORE NEWQUANT
                RETURN

DISPQUANT:      ADD NEWQUANT QUANTITY QUANTITY
                REFILE QONHAND USING PARTNO
                ON=IOERR ERRMSG
                DISPLAY QUANTITY
                RETURN

NEXTPART:       STORE NEWSCREEN
                RETURN

ERRMSG:         MESSAGE I/O ERROR.  CALL SYSTEM MANAGER.
                QUIT

FINISH
```

Figure 5-7. The Program QUPDATE.UP

File Manipulation Statements in QUPDATE.UP

You always need two file manipulation statements in a program that accesses an existing record -- one to bring the record into the program and one to put it back into the database. To bring a record into a program, use a form of the FIND statement. In the routine labeled DISPLAYNAME, we have the statement

```
FIND AND HOLD THE QONHAND USING PARTNO
```

The keyword HOLD locks the record; this prevents another program from accessing it while your program is using it. Use the FIND AND HOLD statement whenever you modify any part of a record.

To replace a record in the database, use the REFILE statement. In the DISPQUANT routine we use the statement

```
REFILE QONHAND USING PARTNO
```

where PARTNO is the key. The REFILE also unlocks the record.

End of Chapter

Chapter 6
Compiling the IFPL Program

To compile an IFPL program and its format, give this command from the CLI:

SYNTAX *[/L] [/A] [/W] [/N]* formatname programname

where:

formatname is the name of a valid format in the current directory.

programname is the name of an IFPL program that exists on your disk. If you use formatname.UP as your **programname**, you don't have to include **programname** in the command line.

The following command switches are optional:

/L Gives you a line printer listing of the source text.

/A Gives you a line printer listing of the source text plus a listing of the assembly language statements that the compiler generates.

/W Suppresses nonfatal error messages; we recommend using it only after inital syntaxing.

/N Compiles the program, but doesn't assemble or load it. It also displays error messages on the terminal screen.

For example,

SYNTAX/L MYPROG

compiles, assembles, and binds the program MYPROG.UP with the format MYPROG. It also sends a source listing to the line printer.

You can also use this form of the SYNTAX command:

SYNTAX listfilename { /L / /A } formatname programname

where:

listfilename is where you want your source and/or assembly listing to go instead of to the line printer. Note that you must use a local **/L** or **/A** switch with the **listfilename**.

For example,

SYNTAX MYLIST/L MYPROG

compiles, assembles, and binds the program MYPROG.UP with the format MYPROG. It also sends a source listing to file named MYLIST instead of to the line printer.

How the Compiler Works

When you give the SYNTAX command, the IFPL compiler goes through this sequence:

1. syntactical phase
2. assembly phase
3. link phase
4. Idea monitor loader phase

In the syntactical phase, the compiler outputs an assembly language version of the source program, named IFPL.SR, where SR stands for source.

In the assembly phase, the assembler uses IFPL.SR to create an object version of the program, named IFPL.OB.

Next, SYNTAX calls the AOS Link, which outputs the program IFPL.PR (for program). IFPL.PR is not executable.

Next, the format loader program, FPYUP, produces the executable program formatname.FP, where the extension .FP stands for field program.

The .FP program is the only one of the intermediate programs that the system retains; it deletes the others. At runtime, the monitor displays literal data on the screen using the file formatname.FS. It then loads formatname.FP. IFMT uses the file named formatname only to display the existing format.

To summarize, we list the following files and their descriptions:

File	Description
formatname	A file describing the visible, terminal screen image format.
formatname.VS	A file containing an evaluation of the format's data fields. (The monitor uses this file to determine field sequence, attributes, and characteristics.)
formatname.FS	A file containing an evaluation of the format's literals.

Note that you must set the user search list to include these files, and you must correctly set the files' access control lists (ACLs).

End of Chapter

Chapter 7
IFPL Statements

This chapter contains alphabetically listed descriptions of the IFPL statements. Table 7-1 lists the statements, their syntax, and their acceptable abbreviations.

Table 7-1. IFPL Statement Summary

Statement	Syntax	Abbreviation
ACCEPT	ACCEPT recordname	
ADD	ADD addend$_1$ addend$_2$ sum	
COMPARE	COMPARE variable$_1$ variable$_2$	COMP
COPY	COMP filename	
DEFINE SUBINDEX	DEFINE subindex USING key...	
DESTROY	DESTROY *[THE]* recordname USING key...	DEST
DISPLAY	DISPLAY { variable / tablename (pointer) }	DISP
DIVIDE	DIVIDE dividend divisor quotient	DIV
DUPLICATES	DUPLICATES *[ARE]* COUNTED *[IN]* variable	DUPL
ENDSUB	ENDSUB	
ENDTABLE	ENTABLE	
ESTABLISH LINK	ESTABLISH LINK *[IN]* filename *[TO]* key...	
FILE	FILE[S] filename$_1$ *[filename$_2$ [filename$_3$]]*	
FILE-NEW	FILE-NEW *[THE]* recordname USING key...	
FIND BEGINNING	FIND *[THE]* recordname BEGINNING *[WITH]* key...	
FIND HOLD	FIND *[AND]* HOLD find-statement	
FIND NEAREST	FIND *[THE]* recordname NEAREST key...	
FIND NEXT	FIND *[THE]* NEXT recordname	

Table 7-1. IFPL Statement Summary (continued)

Statement	Syntax	Abbreviation
FIND PREVIOUS	FIND *[THE]* PREVIOUS recordname	
FIND USING	FIND *[THE]* recordname USING key...	
FINISH	FINISH	FINI
GO TO	GO *[TO]* tag	
GO TO USING	GO *[TO]* tag_1 ,...tag_n USING variable	
[IF] EQUAL	*[IF]* EQUAL tag	
[IF] FOUND	*[IF]* FOUND tag	
[IF] GREATER	*[IF]* GREATER tag	
[IF] IN-RANGE	*[IF]* IN-RANGE tag	
[IF] LESS	*[IF]* LESS tag	
[IF] NOT-EQUAL	*[IF]* NOT-EQUAL tag	
[IF] NOT-FOUND	*[IF]* NOT-FOUND tag	
[IF] OUT-RANGE	*[IF]* OUT-RANGE tag	
INACTIVITY	INACTIVITY CONSTANT *[IS]* value	
INCLUDES	INCLUDES field startingposition length type	INCL
INITIATE PRINTING	INITIATE PRINTING USING printformatname	
IN-RANGE	*[IF]* IN-RANGE tag	
INVERT	INVERT recordname USING key...	
KEY	KEY *[FOR]* { filename / subindexname } *[IS]* length type	
LEFT	LEFT *[JUSTIFY]* $variable_1$ *[IN]* $variable_2$	
LENGTH	LENGTH *[IS]* length	LEN
LESS	*[IF]* LESS tag	
LINK	LINK USING variable *[RETAIN* $file_1$ *[*$file_2$ *] [*$file_3$ *]]*	
LOG	LOG *[THE]* recordname	

Table 7-1. IFPL Statement Summary (continued)

Statement	Syntax	Abbreviation
LOOKUP	LOOKUP *[IN]* tablename (pointer) variable	
MESSAGE	MESSAGE textstring	
MOVE	MOVE { variable$_1$ / tablename$_1$ (pointer) } *[TO]* { variable$_2$ / tablename$_2$ (pointer) }	
MULTIPLY	MULTIPLY multiplicand multiplier product	MUL
NAME	NAME programname	
NODE SIZE	NODE SIZE *[IS]* value	
[ON] BACKTAB	*[ON]* BACKTAB tag	
[ON] DISCONNECT	*[ON]* DISCONNECT tag	
[ON] END DATA	*[ON]* END *[OF]* DATA tag	
[ON] ESCAPE	*[ON]* ESCAPE tag	
[ON] FUNCTION	*[ON]* FUNCTION tag	
ON-IOERR	ON-IOERR tag	ON-IO
[ON] LINE-ERR	*[ON]* LINE-ERR tag	
[ON] LOGOFF	*[ON]* LOGOFF tag	
[ON] MODE CHANGE	*[ON]* MODE CHANGE tag	
[ON] NO-ACTIVITY	*[ON]* NO-ACTIVITY tag	
ON-OVERFLOW	ON-OVERFLOW tag	
[ON] REPEAT	*[ON]* REPEAT tag	
[ON] SCREEN	*[ON]* SCREEN *[IMAGE]* tag	
OUT-RANGE	*[IF]* OUT-RANGE tag	
PARAMETERS FOR SUBINDEX	PARAMETERS *[FOR]* subindexname	
PARTIAL LENGTH	PARTIAL LENGTH *[IS]* value	
PASS	PASS recordname	

Table 7-1. IFPL Statement Summary (continued)

Statement	Syntax	Abbreviation
PERFORM	PERFORM subroutinename	
PRINT	PRINT *[THE]* recordname USING printformatname	
PRIORITY	PRIORITY *[IS]* value	
PROCESS	*[label #*] PROCESS { FILLER / variable } *[AT]* { tag_1 *[AND]* NONE / NONE *[AND]* tag_2 }	PROC
QUEUE	QUEUE variable	
QUIT	QUIT	
RANGE	RANGE $variable_1$ $variable_2$ $variable_3$	
RECEIVE	RECEIVE recordname *[FROM] [ipc-port-name]*	
RECORD	RECORD *[FOR]* { filename / subindexname } *[IS]* recordname	RECD
RECORD FOR PASSING	RECORD *[FOR]* PASSING *[IS]* recordname	
RECORD FOR PRINTING	RECORD *[FOR]* PRINTING *[IS]* recordname	
RECORD FOR TAPE	RECORD *[FOR]* TAPE *[IS]* recordname	
REDEFINES	REDEFINES recordname	
REDESIGNATE	REDESIGNATE register	
REFILE	REFILE *[THE]* recordname USING key...	
REGISTER	REGISTER variable picture *[initial-value]*	REG
REINSTATE	REINSTATE *[THE]* recordname USING key...	
RELEASE	RELEASE { *[THE]* recordname USING key... / ALL HOLDS *[IN]* filename }	
REMOVE	REMOVE *[THE]* recordname	
RESET	RESET { field / label } number	
RESET USING	RESET USING variable	
RETRIEVE HIGH KEY	RETRIEVE HIGH KEY *[FOR]* recordname *[TO]* variable	

Table 7-1. IFPL Statement Summary (continued)

Statement	Syntax	Abbreviation
RETRIEVE KEY	RETRIEVE KEY *[FOR]* recordname *[TO]* variable	
RESTART	RESTART	
RETURN	RETURN { *[file number]* / *[label]* }	RET
RETURN USING	RETURN USING variable	
RIGHT	RIGHT *[JUSTIFY]* variable$_1$ *[IN]* variable$_2$	
SEND	SEND { recordname *[[TO] ipc-port-name]* / REQUEST recordname }	
STOP	STOP	
STORE	STORE variable	
SUBINDEX	SUBINDEX *[FOR]* { filename / subindexname$_1$ } *[IS]* subindexname$_2$	SBIX
SUBROUTINE	SUBROUTINE name	
SUBTRACT	SUBTRACT subtrahend minuend difference	SUB
TABLE	TABLE name	
TERMINATE PRINTING	TERMINATE PRINTING USING printformatname	
VERIFY	VERIFY *[THE]* recordname USING key...	
VERIFY NEXT	VERIFY *[THE]* NEXT recordname	
VERIFY PREVIOUS	VERIFY *[THE]* PREVIOUS recordname	

ACCEPT

ACCEPT recordname

The ACCEPT statement reads a record from the COMMON file into the program variable. To use ACCEPT, you must have defined the record in a RECORD FOR PASSING statement, and you must have sent data to the COMMON file by using a PASS statement.

In the following example, we will pass a record named PARAMETERS into the COMMON file from the program named PROGRAM1. Then, we will use an ACCEPT statement in the program named PROGRAM3 to read the record from the COMMON file.

Note in Figure 7-1 that we didn't give as many INCLUDES statements in the accepting program as in the passing program; you can use only the part of the record that you want. Also, notice that both programs require a RECORD FOR PASSING statement.

```
NAME PROGRAM1            *PASSING PROGRAM
                         .
                         .
                         .
RECORD FOR PASSING IS PARAMETERS
        LENGTH IS 40
        INCLUDES NAME 1 20 ASCII
        INCLUDES ACCOUNTING 21 6 ASCII
        STOP
                         .
                         .
                         .
PASS PARAMETERS
                         .
                         .
                         .
LINK USING PROGRAM3
                         .
                         .
                         .
FINISH
*************************************************************************
NAME PROGRAM3            *ACCEPTING PROGRAM
                         .
                         .
                         .
RECORD FOR PASSING IS PASSREC
                  LENGTH IS 40
                  INCLUDES NAME 1 20 ASCII
                  STOP
                         .
                         .
                         .
ACCEPT PASSREC           *FIRST 20 CHARACTERS
                         *PASSED BY PROGRAM1
                         *ARE NOW AVAILABLE
                         *TO PROGRAM3 IN THE
                         *VARIABLE NAME
```

Figure 7-1. Passing and Accepting Programs

ADD

ADD addend$_1$ addend$_2$ sum

This statement adds **addend$_1$** and **addend$_2$**, placing the result in **sum**. It does not change the values of the addends themselves.

When you define the variable you will use for **sum**, be careful to include enough digits on both sides of the decimal point. The ADD statement first aligns the decimal point of the result. It then rounds and truncates the decimal fraction, and truncates the integer values from the left, if necessary.

If, for example your **sum** variable has a picture 99.99 and your answer was 3333.8775, your **sum** variable would become 33.88.

To ensure that you don't lose valuable digits, give the **sum** variable one more integer place than the larger of the two addends.

COMPARE

COMPARE variable$_1$ variable$_2$

The COMPARE statement compares the value of **variable$_1$** to the value of **variable$_2$** and sets the EQUAL, NOT EQUAL, LESS, or GREATER flag or flags according to the result. You then use the IF EQUAL, IF NOT-EQUAL, IF LESS, or IF GREATER statements to branch to a routine according to the outcome.

NOTE: The flag stays set until the next COMPARE statement.

The COMPARE statement operates with three types of comparison: numeric, alphanumeric, and dissimilar.

Numeric Comparison

If both variables are numeric, COMPARE performs a numeric comparison. For example:

Contents of variable$_1$	Contents of variable$_2$	Flag Set
100.000	0100	EQUAL
746	98.5412	GREATER (and NOT-EQUAL)
085.001	88	LESS (and NOT-EQUAL)

Alphanumeric Comparison

If both variables are alphanumeric, COMPARE first checks their lengths. The longer variable is greater regardless of content. For example:

Contents of variable$_1$	Contents of variable$_2$	Flag Set
SHORT	LONGER	LESS (and NOT-EQUAL)
SHORT	LONG	GREATER (and NOT-EQUAL)

If the two fields are of equal length, COMPARE performs a character-by-character comparison. The letter A is the alphabetic character with the lowest value, and the letter Z has the greatest. Numbers have smaller values than letters. (The comparison is by the ASCII code of the character.)

For example:

Contents of variable$_1$	Contents of variable$_2$	Flag Set
UNIT	UNIT	EQUAL
5347	PRICE	LESS (and NOT-EQUAL)
BTAG	ATAG	GREATER (and NOT-EQUAL)

Dissimilar Comparison

If you compare two variables of dissimilar data type, the compiler issues warning error message, unless you are performing a table comparison. Next, it performs an alphanumeric comparison.

In the case of table comparisons, the compiler assumes that you know the data types of the elements involved, so it won't issue a warning.

COPY

COPY filename

This statement copies the contents of the specified file into your program. To copy a block of statements, place the COPY statement wherever you want the block of statements to appear.

Use the COPY statement when you have several programs that use an identical sequence of statements -- a record definition block, for instance. Since the compiler ignores record field INCLUDES statements that the program doesn't need, you can set up one COPY file and use it in different programs that require different record fields, without tailoring it to each one.

You may nest up to four COPY statements.

DEFINE SUBINDEX

DEFINE subindex USING key...

Use this statement to define a new subindex below the one that the specified key path points to. For example, suppose a file has three index levels -- the root node and two subindex levels -- as in Figure 7-2.

To explicitly define the second subindex level, we would use this statement:

```
DEFINE SUB2 USING AKEY, BKEY
```

Use the DEFINE SUBINDEX statement with the PARAMETERS FOR SUBINDEX block to explicity define subindex parameters.

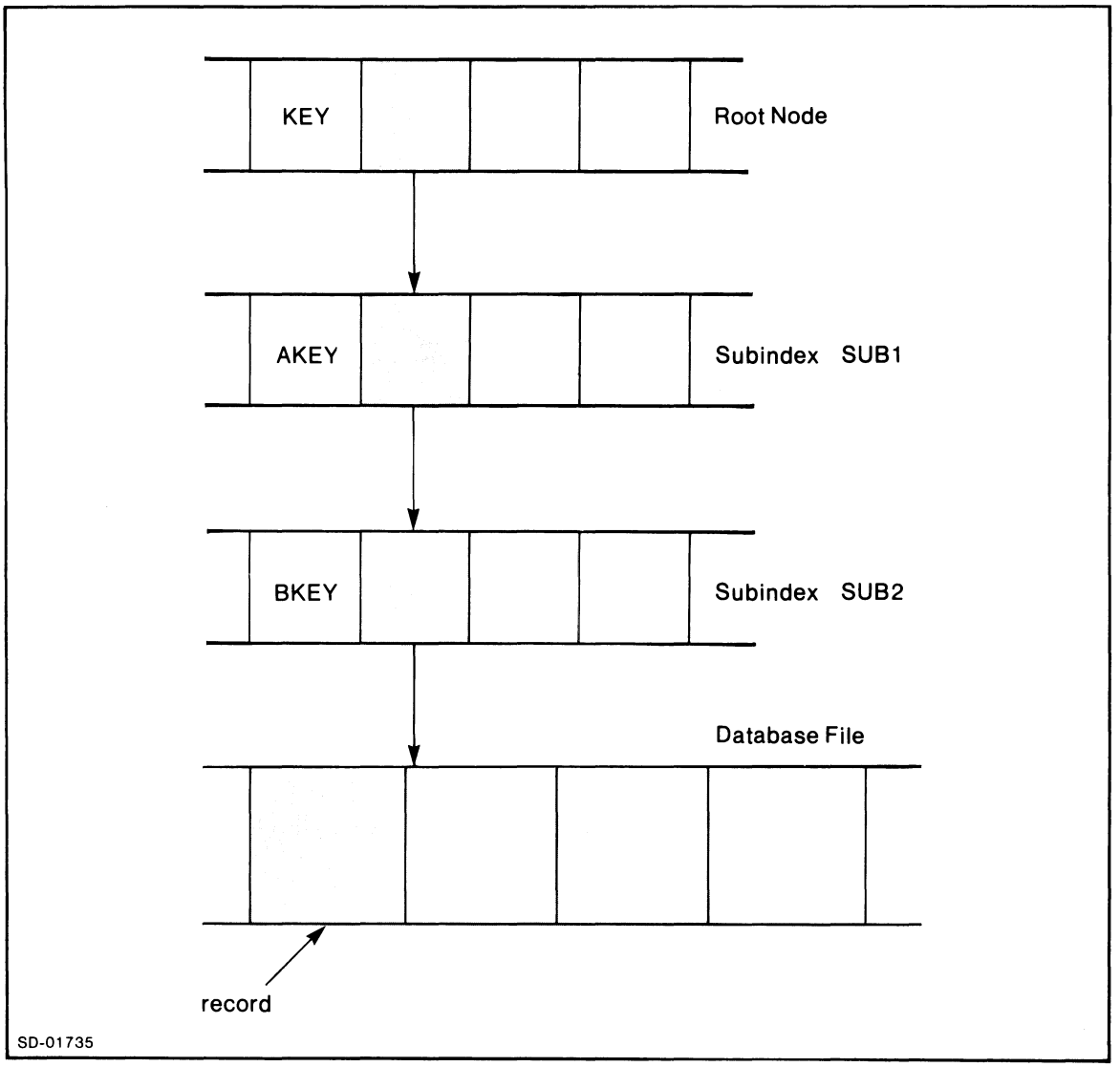


Figure 7-2. A File with Three Index Levels

DESTROY

DESTROY *[THE]* recordname USING key...

This statement physically deletes the specified record. You must use one DESTROY statement for each key. If you have a structure such as

```
AKEY
  |
BKEY
  |
CKEY
```

you must destroy the structure from the bottom up. You cannot delete the entire structure by destroying AKEY; you must first destroy CKEY, then BKEY, then AKEY.

DISPLAY

DISPLAY { variable | tablename (pointer) }

The DISPLAY variable statement displays the current value of **variable** in the current screen format DISPLAY field. You must have given the IFMT DISPLAY attribute to the field. If you try to display a value in an EDIT-only field, the results are unpredictable. Also, you must have previously declared **variable** in a PROCESS or REGISTER statement or assigned a literal variable value to it.

The DISPLAY tablename (pointer) statement displays a value indexed by (pointer) from the table tablename. For example, suppose we have the following table:

```
TABLE ERRORCODES
"00"
"10"
"22"
"23"
END TABLE
```

If the program gives the value 3 to the pointer MPTR, the following DISPLAY statement would display 22:

```
DISPLAY ERRORCODES (MPTR)
```

The DISPLAY will occur when the program executes the RETURN statement associated with the routine. You can display only one field per field-processing routine.

If you attempt to display a value that exceeds the DISPLAY field's specification, the monitor will display a field of asterisks.

DIVIDE

DIVIDE dividend divisor quotient

This statement divides the value of **dividend** by the value of **divisor** and places the result in the variable **quotient**.

To ensure that you don't truncate quotient digits, declare your **quotient** variable with as many integer digits as **dividend** and as many decimal places as **divisor**.

DUPLICATES

DUPLICATES *[ARE]* COUNTED *[IN]* variable

Use this statement with files allowing duplicate keys. Place the DUPLICATES statement immediately after the KEY statement of the subindex allowing duplicate keys.

You must use a REGISTER statement within the program to declare **variable** as a numeric variable, or use a PROCESS statement to associate it with a numeric field on the screen.

When the program uses a FILE-NEW, a FIND NEAREST, or a FIND BEGINNING statement, the compiler places the duplicate count in **variable**.

ENDSUB

ENDSUB

Place this statement at the end of all subroutines. ENDSUB tells the compiler where the end of the subroutine is. It also returns program control to the statement following the PERFORM statement that called the subroutine.

ENDTABLE

ENDTABLE

This must be the last statement in a table definition. It tells the compiler where the end of the table is.

ESTABLISH LINK

ESTABLISH LINK *[IN]* filename *[TO]* key...

Use this statement to create alternate key paths to records.

You must first create the keys you wish to use in the ESTABLISH LINK statement. One way to do this is to define dummy records that use these keys in FILE-NEW statements.

Next, you must position the INFOS system pointer to the level of the existing key path where you want to create the alternate path. Use a FIND or VERIFY statement to access a record on that level. This can also be a dummy record, as long as it is on the correct level.

Next, use the ESTABLISH LINK statement to create the alternate key path from that level.

For example, Figure 7-3 shows an index structure with a link. The file is a customer database. The first index level is for region, the second is for customer name, and the third is for invoice number. The link we create will use a customer number as a key; through this link we can access an invoice record by knowing just the customer number and the invoice number.

Figure 7-4 shows the program that will establish the customer number link into the index structure. Note the use of dummy records; we use them to position to the proper level.

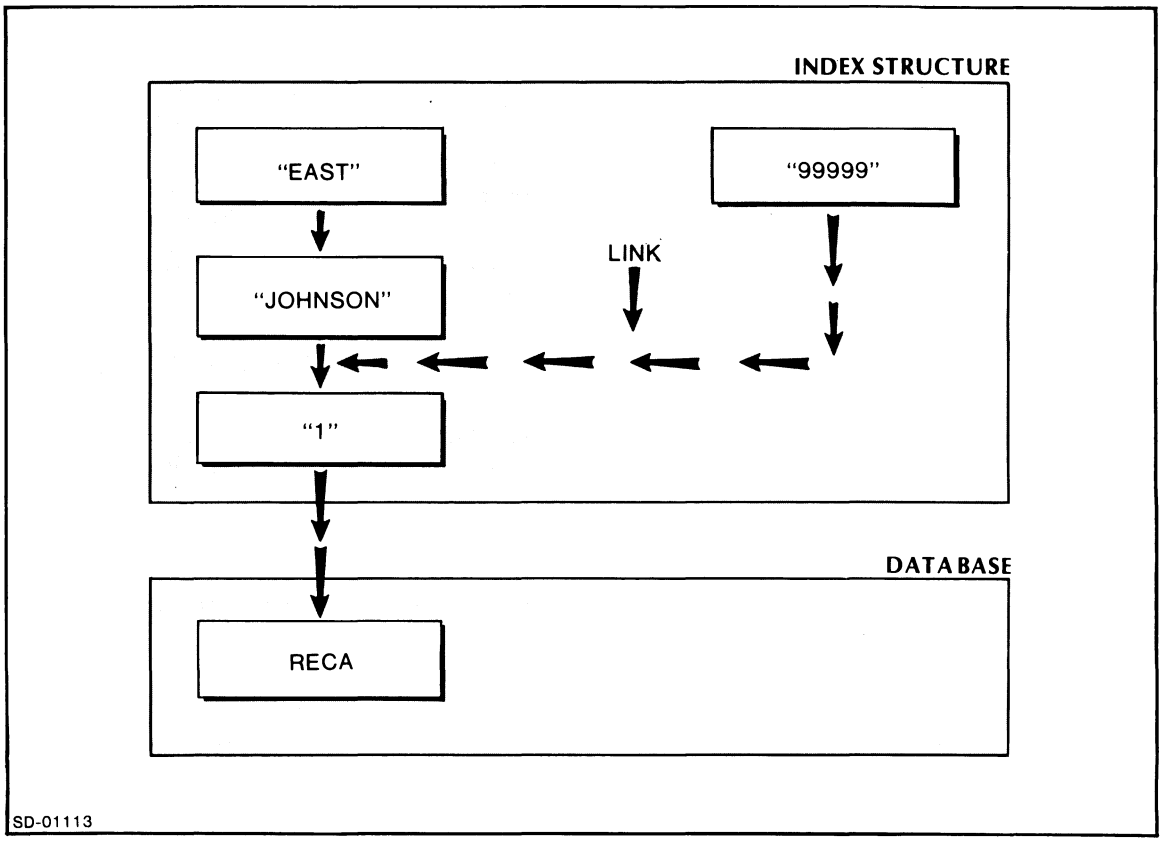


Figure 7-3. An Index Structure with a Link Between a Key Sequence and a Subindex

```
        NAME LINKTEST
        FILE NFILE
        SUBINDEX FOR NFILE IS LEVEL1
        SUBINDEX FOR LEVEL1 IS LEVEL2
        KEY FOR NFILE IS 13 ASCII
        KEY FOR LEVEL1 IS 13 ASCII
        KEY FOR LEVEL2 IS 6 ASCII
*THE RECORD THAT FOLLOWS (LEVELOREC) IS A
*DUMMY RECORD USED TO WRITE KEY "99999"
        RECORD FOR NFILE IS LEVELOREC
                LENGTH IS 0
                STOP
*THE RECORD THAT FOLLOWS (LEVEL1REC) IS A
*DUMMY RECORD USED TO POSITION TO
*KEY "JOHNSON"
        RECORD FOR LEVEL1 IS LEVEL1REC
                LENGTH IS 0
                STOP
        RECORD FOR LEVEL2 IS LEVEL2REC
                LENGTH IS 80
                INCLUDES FIELD2 1 80
                STOP
        REGISTER FIELD2 X(80)
        PROCESS FILLER AT D1 AND NONE
D1:
*CREATE INTIAL RECORD
        FILE-NEW LEVEL2REC USING "EAST", "JOHNSON" AND "1"
*CREATE THE UPPER LEVEL OF THE SECOND KEY PATH
        FILE-NEW LEVELOREC USING "99999"
*POSITION TO SUBINDEX TO BE LINKED TO NOTE
*THAT THE POSITION IS ABOVE THE KEY TO BE
*USED IN THE NEW PATH
        VERIFY LEVEL1REC USING "EAST"  AND "JOHNSON"
*CREATE THE LINK
        ESTABLISH LINK IN NFILE TO "99999"
*TRY OUT THE NEW KEY PATH
        FIND LEVEL1REC USING "99999" AND "1"
        ON-IOERR D1A
        MESSAGE LINK SUBINDEX SUCCESSFUL
        QUIT

D1A:

        MESSAGE LINK SUBINDEX UNSUCCESSFUL
        QUIT
        FINISH
```

Figure 7-4. Using ESTABLISH LINK to Create an Index Structure

In the program, we created a dummy record (with length 0) so that we could use the key 99999 in a FILE-NEW statement, thus creating the key. We also used the VERIFY statement to position to level 1.

The keys you use in the ESTABLISH LINK statement must describe a complete index path; they cannot contain subindexes. However, the position to which you are linking must have a subindex below it. Therefore, in our example, we could not link to the record directly; we had to link at the level above the last key.

You may use any pathway to access any record, regardless of which pathway you used to create it. For example, suppose you have 100 invoice records for a customer: 50 that you created with the path EAST,JOHNSON,n (where *n* is the invoice number), and 50 that you created with 99999,n (where *99999* is the Johnson Company's number). You can then access any of the records using either of the paths.

Of course, you need to establish a separate link for each customer in the file. WEST,SMITH,n will have its own customer number link, such as 99998,n. Also, the new key path may be shorter, the same length, or longer than the original path.

The ESTABLISH LINK statement can save you space. Consider a file that has items filed under NAME, ACCOUNT, LINEITEMS, and inverted under REGION, ACCOUNT, LINEITEMS. This creates the large duplicate index structure shown in Figure 7-5.

An ESTABLISH LINK statement can create the structure shown in Figure 7-6, which avoids the unnecessary overhead of Figure 7-5.

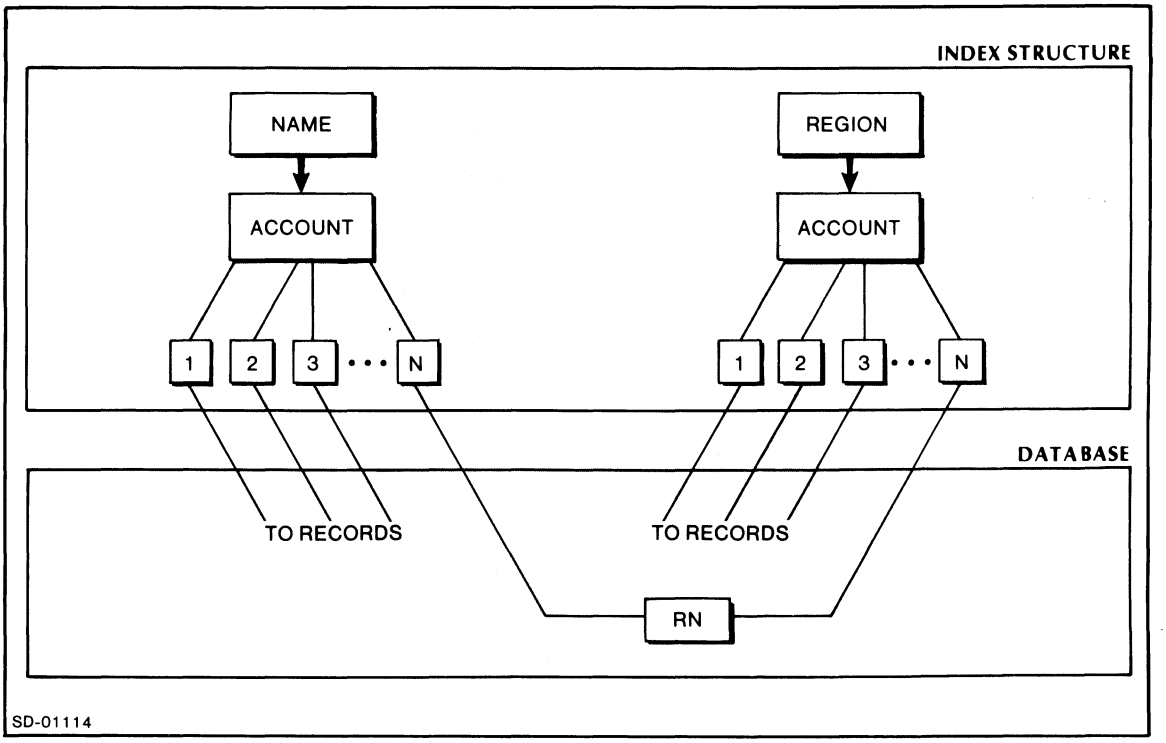


Figure 7-5. A File with Inverted Database Records and Unnecessarily Duplicated Subindexes

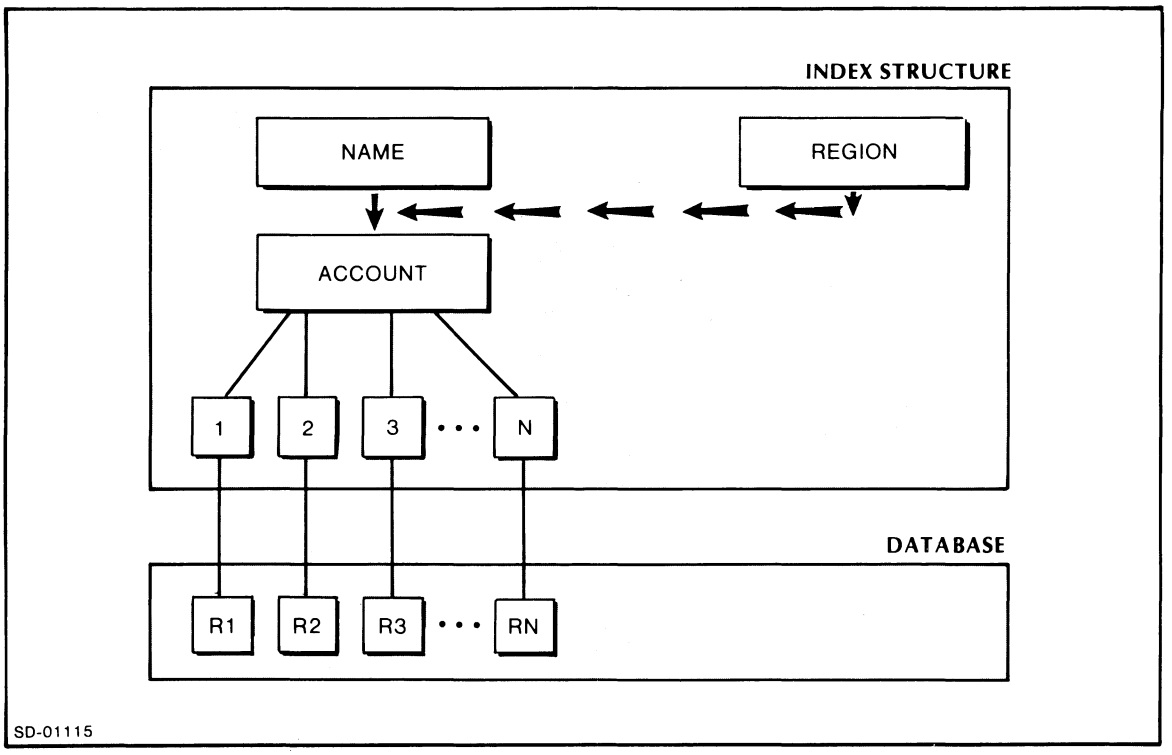


Figure 7-6. Figure 7-5 Reconfigured Using ESTABLISH LINK

FILE

FILE[S] filename$_1$ *[filename$_2$ [filename$_3$]]*

This statement tells the compiler which files the program will use. You may specify a maximum of three files. You must have previously created the files with ICREATE or with a COBOL program.

Pathnames must consist of 14 or fewer characters.

FILE-NEW

FILE-NEW *[THE]* recordname USING key...

Use the FILE-NEW statement to write new records into a file. You must define **recordname** in a record definition block within the program.

For example, the program named INITDEP.UP in Figure 7-7, initializes a file record that keeps a bank customer's balance; the **key** is the account number.

```
NAME INITDEP
FILE BALANCE
        KEY FOR BALANCE IS 4 ASCII
        RECORD FOR BALANCE IS BALREC
                LENGTH IS 10
                INCLUDES OLDBAL 1 10 ASCII
        STOP

PROCESS ACCOUNT AT NONE AND GETACCOUNT
PROCESS OLDBAL AT NONE AND GETBAL

GETACCOUNT:     STORE ACCOUNT

                RETURN

GETINITDEP:     STORE OLDBAL
                FILE-NEW BALREC USING ACCOUNT
                ON-IOERR ERRMSG
                MESSAGE RECORD ADDED TO DATABASE

                RETURN 1

ERRMSG:         MESSAGE ACCOUNT ALREADY ON FILE
                RETURN 1

FINISH
```

Figure 7-7. FILE-NEW Example

FIND BEGINNING

FIND *[THE]* recordname BEGINNING *[WITH]* key...

The FIND BEGINNING statement retrieves the record **recordname** using a generic (partial) **key**. You must specify one key for each level of subindexes. However, the last key in the list is the one that the system uses as a generic key to search for the record.

For example, suppose you have a two-level index. The key for the first level is ACCTNO; the key for the second level is NAME.

```
KEY FOR LEVEL1 IS 5 ASCII
      .
      .
      .
KEY FOR LEVEL2 IS 10 ASCII
      .
      .
      .
FIND THE CUSTREC BEGINNING WITH ACCTNO,NAME
```

ACCTNO takes you through the first level; it must be an exact key.

NAME searches the second level for a key beginning with whatever is in the NAME field. For example, if NAME has the value SM and you have records stored under the names SMITH and SMYTH, the FIND BEGINNING statement will retrieve SMITH's record.

FIND BEGINNING uses the input length of the key as its length; it doesn't use the length specified for the key in the KEY statement. Therefore, in our example, the key NAME is two bytes long (SM), even though the KEY statement says that it's 10 bytes.

FIND HOLD

FIND *[AND]* HOLD find-statement

You may use the phrase *[AND]* **HOLD** in any FIND statement. The **HOLD** keyword locks the record against access by any other program.

To update a locked record and free it for use by another program, use the REFILE statement. To free the record for access by other programs, use the RELEASE statement.

FIND NEAREST

FIND *[THE]* recordname NEAREST key...

FIND NEAREST retrieves the record **recordname** by using an approximate **key**. The approximate key must have an ASCII value less than or equal to the key you're looking for.

For example, suppose your records are keyed by PONUMB, and you have two records with the keys 21 and 700, respectively. If you give PONUMB the value 22, and then give this statement:

```
FIND THE CASHREC NEAREST PONUMB
```

you will access the record with the key 700.

If the approximate key happens to hit an actual key, the statement will access that key's record.

FIND NEXT

FIND *[THE]* NEXT recordname

This statement lets you process a database sequentially. First you use a FIND USING, FIND BEGINNING, or FIND NEAREST statement to position yourself within the database. You can then use FIND NEXT to retrieve the record immediately following the current one.

For example,

```
FIND THE CREC USING MASTNO, CUSTNAM
      .
      .
      .
FIND THE NEXT CREC
```

If MASTNO contains 20 and CUSTNAM contains TAYLOR, the FIND USING statement will retrieve the record keyed by 20, TAYLOR. If the database contains records with the keys 20,JOHNSON; 20,TAYLOR; 20,ZONIS,; then the FIND NEXT statement will retrieve ZONIS's record.

FIND PREVIOUS

FIND *[THE]* PREVIOUS recordname

Use this statement after a FIND USING, FIND BEGINNING, or FIND NEAREST statement to scan backwards through the database. For example, given the following statements:

```
FIND THE AREC USING CUSTNO
      .
      .
      .
FIND THE PREVIOUS AREC
```

If CUSTNO has the value 38 and the database has records with keys 17, 38, and 40, then the FIND USING statement will access the record with key 38. The FIND PREVIOUS statement will access the record with key 17.

FIND USING

FIND *[THE]* recordname USING key...

This is the primary data-retrieval statement. The INFOS system will locate and retrieve the record with the specified key(s).

You may use as many as 15 keys with this statement, and you must use one key for each index level you wish to traverse. The keys cannot be longer than the length specified in the KEY statement. If you have used binary or packed keys, the system will convert them to ASCII values before using them. The system will also convert binary or packed record information.

FINISH

FINISH

This must be the final statement in every IFPL program. It tells the compiler that it has reached the end of the program. A FINISH statement must be the last statement, even if the program ends somewhere else with a QUIT statement.

GO TO

GO *[TO]* tag

This is an unconditional GO TO statement; it directs program execution to the routine labeled **tag**.

GO TO USING

GO *[TO]* tag_1, ...tag_n USING variable

This is a conditional branching statement. The system checks the contents of **variable**, which must be numeric. Its value determines which **tag** the program will branch to. If **variable** has the value 1, the program will branch to the routine labeled by the first tag; if **variable** has the value 25, the program will branch to the routine labeled by the 25th tag. You can have 40 arguments with an IFPL statement, which means you can include 38 tags in a GO TO USING statement. (USING and **variable** are the other two arguments.)

If **variable** has a value less than 1 or greater than the number of tags you've specified, program control steps to the next program statement.

IF EQUAL

[IF] EQUAL tag

This statement checks the EQUAL flag set by the most recent COMPARE statement. If the flag is set (meaning that the two COMPAREd values were equal), the EQUAL statement sends program control to the statement labeled by **tag**.

IF FOUND

[IF] FOUND tag

Use this statement in conjunction with the LOOKUP statement. If the latest LOOKUP statement succeeded in finding the table element it was searching for, the compiler sets the flag accordingly, and the IF FOUND statement will send the program to the routine labeled by **tag**. Figure 7-8 gives an example.

```
NAME REORDER
FILE IS INVENTORY
KEY FOR INVENTORY IS 4 ASCII
RECORD FOR INVENTORY IS INVREC
        LENGTH IS 20
        INCLUDES PARTNAME 1 20 ASCII
STOP

PROCESS PARTNO AT NONE AND GETPARTNO
PROCESS PARTNAME AT DISPLAYNAME AND NONE
PROCESS DEPTNO AT DISPLDEPT AND NONE

TABLE DEPTA
"C330"
"S130"
"CS40"
ENDTABLE

TABLE DEPTB
"X250"
"Y930"
"Z280"
ENDTABLE

TABLE DEPTD
"CS30"
"M600"
"E500"
ENDTABLE

GETPARTNO:      STORE PARTNO
                RETURN

DISPLAYNAME:    FIND THE INVREC USING PARTNO
                ON-IOERR ERRMSG
                DISPLAY PARTNAME
                REFILE INVREC USING PARTNO
                ON-IOERR ERRMSG
                RETURN

DISPLDEPT:      LOOKUP IN DEPTA PARTNO
                IF FOUND D1
                LOOKUP IN DEPTB PARTNO
                IF FOUND D2
                LOOKUP IN DEPTD PARTNO
                IF FOUND D3

D1:             MOVE "DEPT A" TO DEPTNO
                DISPLAY DEPTNO
                RETURN

D2:             MOVE "DEPT B" TO DEPTNO
                DISPLAY DEPTNO
                RETURN

D3:             MOVE "DEPT D" TO DEPTNO
                DISPLAY DEPTNO
                RETURN

ERRMSG:         MESSAGE IO-ERROR.  CALL SYSTEM MANAGER.
                QUIT

FINISH
```

Figure 7-8. The IF FOUND Statements Branch to the Appropriate Routines

IF GREATER

[IF] GREATER tag

This statement checks the flag set by the most recent COMPARE statement. If it's set to GREATER, this statement sends program control to the routine labeled by **tag**.

IF IN-RANGE

[IF] IN-RANGE tag

This statement checks the IN-RANGE flag set by the most recent RANGE statement. If the flag is set, the program branches to the routine labeled by **tag**.

IF LESS

[IF] LESS tag

This statement checks the LESS flag set by the most recent COMPARE statement. If the flag is set, the IF LESS statement sends program execution to the routine labeled by **tag**.

IF NOT-EQUAL

[IF] NOT-EQUAL tag

This statement checks the NOT-EQUAL flag set by the most recent COMPARE statement. If the flag is set, the program branches to the routine labeled by **tag**.

IF NOT-FOUND

[IF] NOT-FOUND **tag**

This statement checks the flag set by the most recent LOOKUP statement. If the flag is set to 0 (meaning that the LOOKUP didn't find the table element), then the program branches to the routine labeled by **tag**.

IF OUT-RANGE

[IF] OUT-RANGE **tag**

The OUT-RANGE statement checks the OUT-RANGE flag set by the most recently executed RANGE statement and branches to **tag** if that flag is set.

INACTIVITY

INACTIVITY CONSTANT *[IS]* **value**

This statement sets the length of time, in minutes, that an IFPL program will wait for the operator to enter data. If the operator doesn't enter data within the specified amount of time, the program takes appropriate action by using the ON NO-ACTIVITY statement. Therefore, if you use the INACTIVITY statement, you must also include an ON NO-ACTIVITY statement. See the ON NO-ACTIVITY statement for more information.

INCLUDES

INCLUDES field startingposition length type

where:

field	is defined elsewhere in the program as a register, a PROCESS variable, or a literal.
startingposition	is the character position within the record where this particular field begins.
length	is the length of the field (in bytes for ASCII or ALPHA).
type	is ASCII (or ALPHA), BINARY, OR PACKED.

You can use the INCLUDES statement only within a record definition block. It identifies significant fields within the record. For example, suppose you have a 15-byte record that contains the information in Figure 7-9.

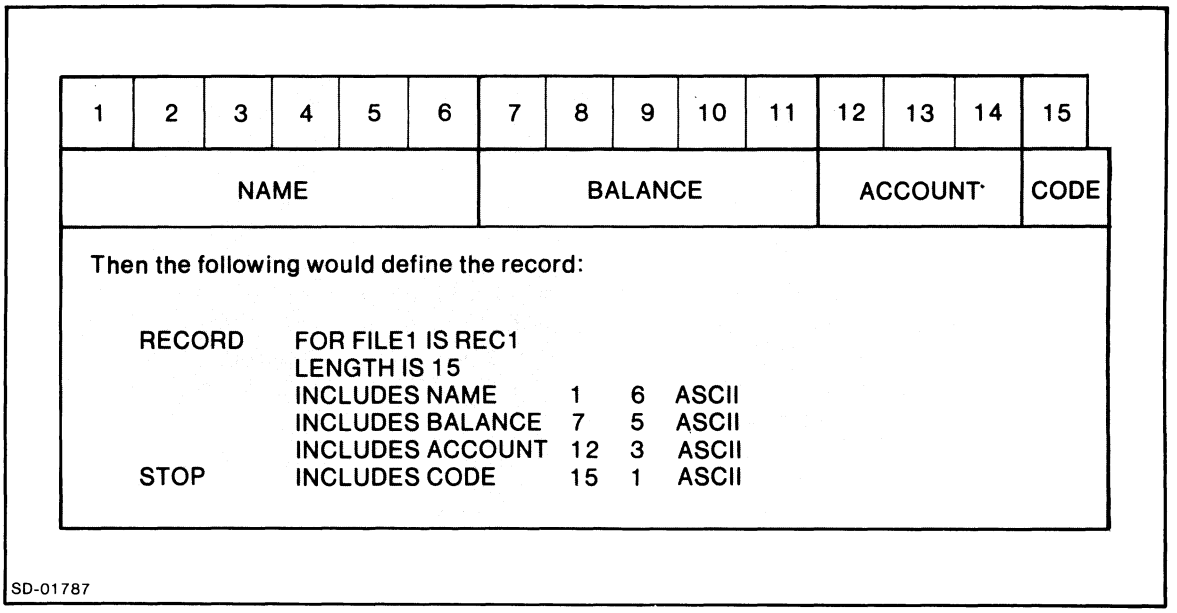


Figure 7-9. INCLUDES Example

When you access a record, you don't have to use all the variables stored within it. Suppose you created a record with a program containing this record definition block:

```
RECORD FOR AFILE IS REC1

LENGTH IS 106

INCLUDES ELEMENTA 1 4 ASCII

INCLUDES ELEMENTB 5 7 ASCII

      .

      .

      .

INCLUDES ELEMENTS 98 9 ASCII

STOP
```

Later, you could access only those record elements that you needed in another program:

```
RECORD FOR AFILE IS REC1

LENGTH IS 106

INCLUDES ELEMENTA 1 4 ASCII

INCLUDES ELEMENTF 5 18 ASCII

INCLUDES ELEMENTQ 23 8 ASCII

STOP
```

Remember that you must define your variables in REGISTER or PROCESS statements, or else use literals. There is one exception: if you use a COPY file to define the record, you do not have to define every field that appears in the record definition block; you just have to define the fields that you want to use.

The compiler expands BINARY or PACKED types to ASCII lengths when it accesses them. Table 7-2 shows the lengths to specify in the INCLUDES statement, and the length to which the system will expand INCLUDES during access.

Table 7-2. BINARY and PACKED INCLUDES

Field Length in INCLUDES Statement Specification	Number of Digits in IFPL Register
BINARY	
1	1-2
2	3-4
3	5-6
4	7-9
5	10-11
6	12-14
7	15-16
PACKED	
1	1
2	2-3
3	4-5
4	6-7
5	8-9
6	10-11
7	12-13
8	14-15
9	16-17
10	18

The sign in a PACKED field requires one-half of a byte; it is stored in the last half-byte. Figure 7-10 shows how the system stores a 5-digit PACKED field.

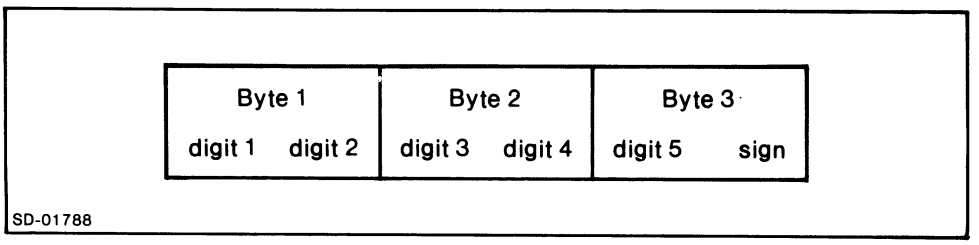


Figure 7-10. A 5-digit PACKED INCLUDES

INITIATE PRINTING

INITIATE PRINTING USING printformatname

This statement marks the beginning of a set of printing records in the COMMON file. You must specify the **printformatname** in subsequent PRINT statements.

After you have built the print file with PRINT statements, you mark the end of it with the TERMINATE PRINTING statement.

INVERT

INVERT recordname USING key...

Use this statement to write an alternative pathway to an existing record. For example, if you have a database that contains customer records keyed by customer number, you can use the INVERT statement to build an index pathway that will access the records by customer name. See Figure 7-11 for an example.

```
FILE AFILE
        KEY FOR AFILE IS 5 ASCII

RECORD FOR AFILE IS AREC
        LENGTH IS 40
        INCLUDES ACCTNO 1 5 ASCII
        INCLUDES NAME 6 12 ASCII
                .
                .
                .
STOP

        .
        .
        .
FILE-NEW AREC USING ACCTNO
INVERT AREC USING NAME
        .
        .
        .
FINISH
```

Figure 7-11. INVERT Example

You will normally use the INVERT statement immediately following a FILE-NEW, FIND, or REFILE statement. INVERT uses an internal pointer set by those three statements, so you cannot put another I/O statement between the FILE-NEW, FIND, or REFILE statement and the INVERT statement.

The LOG statement, however, does not reset the internal pointer. Therefore, you can interpose a LOG statement between the INVERT and FILE-NEW, FIND, or REFILE.

KEY

KEY *[FOR]* {filename / subindexname} *[IS]* length type

where:

length is the length of the key field.

type is either ASCII, BINARY, or PACKED.

This statement defines the key length and type for a file or subindex. You must define the **filename** or the **subindexname** in a FILE or SUBINDEX statement that appears before this statement.

If you use type ASCII, specify the number of characters in the key. The key's register, screen field, or literal will define the actual key length; the KEY statement defines the maximum length of the key. Consequently, to access a record created with a key that is eight bytes long, you must use the full eight bytes. Consider these two program fragments:

```
PROGRAM 1

REGISTER NAME X(8)
STORE NAME

FILE-NEW AREC USING NAME
```

```
PROGRAM 2

REGISTER NAME X(7)
STORE NAME

FIND AREC USING NAME
```

Program 2 will not be able to access the records created by program 1. This would be true even if both programs contained the statement KEY FOR AFILE IS 7 ASCII.

For BINARY or PACKED types, Idea converts the key value in the given variable to the specified type. Table 7-3 shows the number of digits to specify for the key's value.

Table 7-3. BINARY and PACKED Keys

KEY Statement Specification	Size of Variable in Digits
BINARY	
1	1-2
2	3-4
3	5-6
4	7-9
5	10-11
6	12-14
7	15-16
PACKED	
1	1
2	2-3
3	4-5
4	6-7
5	8-9
6	10-11
7	12-13
8	14-15
9	16-17
10	18

The sign in a PACKED key requires one-half of a byte; it is stored in the last half of the byte. Figure 7-12 shows how the system stores a 5-digit PACKED key.

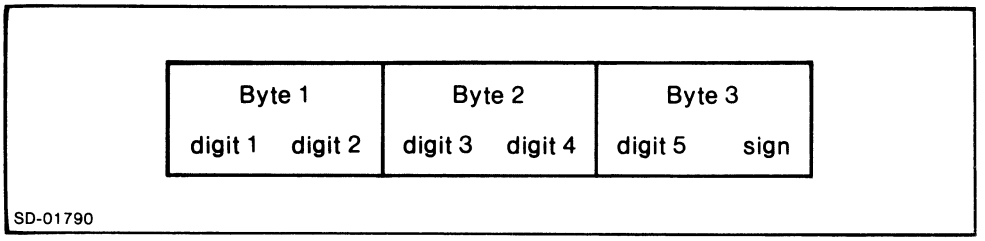


Figure 7-12. A 5-digit PACKED Key

LEFT

LEFT *[JUSTIFY]* variable$_1$ *[IN]* variable$_2$

This statement will left-justify a source field in a larger destination field.

The LEFT statement moves data from variable$_1$ to variable$_2$, starting with the left-most character position in each field and proceeding from left to right. A LEFT move is like an alpha move except that you can use it on any data type.

LEFT treats blanks in a source field like any other character. It performs no zero- or blank-filling in the destination. If the destination is longer than the source, the system will retain the excess destination data.

The system will disregard a decimal point in the source field, but it will display a decimal point in the destination field if you specify one in the field's picture.

The system performs data transfers with fields of matching data types and size on a character-position-by-character-position basis. No justification is involved in such moves since blanks are treated like data.

Table 7-4 shows the results of some example LEFT moves.

Table 7-4. Moving Data with the LEFT Statement

Example Type	Initial Values	Final Dest Values
Numeric Srce<Dest No Decimal Point	Srce=788 Dest=55555	78855
Numeric Srce>Dest No Decimal Point	Srce=83492 Dest=671	834
Numeric Srce>Dest Decimal Point	Srce=16.98 Dest=178.544	169.844
Numeric Srce>Dest Decimal Point	Srce=856.99 Dest=28.5	85.6
Alphanumeric Srce<Dest	Srce=patnum Dest=Vacancy	patnumy
Alphanumeric Srce>Dest	Srce=patnum Dest=Vac	pat
Mixed Srce<Dest	Srce=858.9 Dest=station	8589ion
Mixed Srce>Dest	Srce=sub Dest=6.3	su

Dest = Destination. Srce(Source) remains unchanged.

LENGTH

LENGTH *[IS]* length

where:

length is the length of the record in bytes.

You must place a LENGTH statement after every RECORD statement in a record definition block, unless the REDEFINES statement is the only statement in the record definition block.

Initializing the Record Buffer

To initialize the record buffer to zero or blank, use a dummy INCLUDES statement that is as long as the record.

To blank out a buffer, use the following:

```
RECORD FOR PASSING IS PASSREC
    LENGTH IS 200
    INCLUDES " " 1 200 ASCII
    INCLUDES F1 2 10 ASCII
    INCLUDES F2 12 4 ASCII
        .
        .
        .
    STOP
```

To zero out a buffer, use this:

```
RECORD FOR PASSING IS PASSREC
    LENGTH IS 200
    INCLUDES "0" 1 200 ASCII
    INCLUDES F1 2 10 ASCII
        .
        .
        .
    STOP
```

We recommend that you avoid using literals in records intended to receive data, since literals may change, producing unexpected results.

LINK

LINK USING variable *[RETAIN file$_1$ [file$_2$ [file$_3$]]]*

The LINK statement lets you link to a new format under program control. This is a different means of linking than the IFMT linking facility.

You can link one program to another with both a LINK statement and a linked format created with IFMT; neither affects the other.

The **variable** must be a literal or a variable defined by a REGISTER or PROCESS statement, and it must contain the name of a valid format.

The **RETAIN file$_1$...file$_3$** argument is an optional clause that allows you to continue using the named files across linked programs, without the overhead of closing and then opening the files after linking. See Figure 7-13 for an example.

```
                NAME PROGRAM
                FILE MASTER, INVENTORY
                REGISTER PROG1 XXXXXXXX PROGRAM1
                REGISTER PROG2 XXXXXXXX PROGRAM2
                .
                .
                .
E1:             STORE ANSWER
                COMPARE ANSWER YES
                IF EQUAL LPRG2
                LINK USING PROG1 RETAIN MASTER

LPRG2:          LINK USING PROG2 RETAIN INVENTORY

                *linked program
                NAME PROGRAM1
                FILES UPDATE, MASTER

                .
                .
                .
```

Figure 7-13. The RETAIN Clause Lets You Keep Files Open

The program named PROGRAM links to the format named PROGRAM1, via the statement LINK USING PROG1 RETAIN MASTER. PROGRAM will close the file INVENTORY but keep the file MASTER open for use with PROGRAM1.

LOG

LOG *[THE]* **recordname**

Use the LOG statement to write a record to magnetic tape.

The system sends all tape-logging errors to the special register IOERR, so your tape-logging programs should contain error-handling routines. The error codes sent to IOERR are:

IOERR = 18 Record length longer than the maximum specified with the IDEASG utility.

IOERR = 30 Physical tape error (such as parity).

IOERR = 34 End of tape file.

If any of these conditions occurs, the monitor sends the error code to the reserved word IOERR, the error log (ELOG), and, possibly, the supervisory console.

LOOKUP

LOOKUP *[IN]* **tablename (pointer) variable**

This statement searches a table for a value. If it finds an element whose value is the same as the value of **variable**, it places the index number of that table element in **(pointer)**; otherwise, it sets **(pointer)** to zero.

The index number is **variable's** position in the table. The table's first element is 1, the second is 2, and so forth.

If you don't specify a **(pointer)**, the monitor places the index value in the special register ENTRY. You can use ENTRY anywhere you use a register.

LOOKUP also sets a flag to either 0 or the index number. The FOUND and NOT-FOUND statements branch to routines depending on the flag's value. (See FOUND and NOT-FOUND.)

MESSAGE

MESSAGE textstring

The MESSAGE statement sends a message to the operators' consoles. The **textstring** starts with the first nondelimiter; you should end the text string with a NEW LINE.

You may use any text string, including spaces, up to 80 characters long. Also, you can send special control characters (outside the standard set of alphanumerics) by enclosing the 2-character octal equivalent in angle brackets; e.g., <07>. You may also enclose the bracketed code in exclamation points to disable and re-establish interpretation by the IDEA terminal interface routines.

For example:

```
MESSAGE !<47><57>! THIS IS A MESSAGE
```

To send the contents of a variable as a message, surround the variable with brackets, as in this example:

```
MESSAGE [OLDBAL]
```

The variable must be flush left against the bracket, and it must be the only argument; otherwise, the monitor will send the message verbatim. For example, this message statement

```
MESSAGE OLD BALANCE IS [OLDBAL]
```

will display

```
OLD BALANCE IS [OLDBAL]
```

on the screen.

MOVE

MOVE { variable$_1$ / tablename$_1$ (pointer) } *[TO]* { variable$_2$ / tablename$_2$ (pointer) }

The MOVE statement has the general form of

MOVE source-variable destination-variable

In all MOVEs the value of the **source-variable** replaces the value of the **destination-variable**. (The value of the **source-variable** is unchanged.)

MOVE variable$_1$ TO variable$_2$

copies the value of **variable$_1$** into **variable$_2$**.

MOVE variable$_1$ TO tablename (pointer)

copies the value of **variable$_1$** into the table element referenced by **(pointer)**.

MOVE tablename (pointer) TO variable$_2$

copies the value of the table element referenced by **(pointer)** into **variable$_2$**.

MOVE tablename$_1$ (pointer$_1$) TO tablename$_2$ (pointer$_2$)

copies the value of the **tablename$_1$** element referenced by **(pointer$_1$)** into the **tablename$_2$** element referenced by **(pointer$_2$)**.

The parentheses are part of the command; you must enclose the pointers in parentheses.

The system does not check data types for MOVEs using table elements; it assumes that the source and destination data types are identical.

If you perform a MOVE with a **source-variable** that is shorter than the **destination-variable**, the compiler pads the destination. With MOVEs involving alphabetic or alphanumeric values, it pads the destination from the left with blanks. For numeric MOVEs, Idea aligns the decimal point, then pads from the right and left, as necessary, with zeros.

If you perform a MOVE with dissimilar data types, the compiler issues a warning, performs an alphabetic MOVE, and deletes the decimal point.

Note that the MOVE statement doesn't round; it truncates. See Table 7-5 for examples.

Table 7-5. Parameter-Fitting by the MOVE Statement

Type of MOVE	Initial Values	Final Dest Values
Numeric Srce<Dest No Decimal Point	Srce=788 Dest=55555	00788
Numeric Srce<Dest No Decimal Point	Srce=83492 Dest=671	492
Numeric Srce<Dest Decimal Point	Srce=16.98 Dest=178.544	016.980
Numeric Srce>Dest Decimal Point	Srce=856.99 Dest=28.5	56.9
Alphabetic Srce<Dest	Srce=pathnum Dest=vacancy	patnum
Alphabetic Srce>Dest	Srce=patnum Dest=vac	pat
Mixed Srce<Dest	Srce=858.9 Dest=station	8589 syntax warning
Mixed Srce>Dest	Srce=sub Dest=6.3	su syntax warning

Dest = Destination. Srce (Source) remains unchanged.

MULTIPLY

MULTIPLY mutiplicand multiplier product

The MULTIPLY statement multiplies the contents of **multipicand** by the contents of **multiplier**, and places the result in **product**.

To avoid losing significant integer digits to truncation, give your product variable as many integer digits as the **multipicand** plus the **multipier**.

NAME

NAME programname

The NAME statement assigns a name to your program. It must be the first statement in the program and must be used only once within the program.

There is no logical connection between **programname** and the AOS filename you give to the source text file, but we recommend that you use the same name.

The **programname** must begin with a letter. The remaining characters can be letters, numbers, or periods(.).

Do *not* use the following characters in program names:

dash	-
colon	:
carat	^
single quote	'
double quote	''
angle brackets	< >
parentheses	( )

NODE SIZE

NODE SIZE *[IS]* value

Use this statement within the PARAMETERS FOR SUBINDEX block. NODE SIZE explicitly defines the node size of a subindex. The **value** may be either 2042 or 4090 (bytes). The default value (if you don't use this statement) is 2042.

ON BACKTAB

[ON] BACKTAB tag

Place the ON BACKTAB statement anywhere among the nonexecutable statements except among the PROCESS statements. This statement allows the program to take some action if the operator strikes the BACKTAB key. When this happens, the ON BACKTAB statement transfers program control to the routine labeled tag.

The BACKTAB key is the unlabeled key on the cursor pad.

ON DISCONNECT

[ON] DISCONNECT tag

If the operator's dial-up line becomes disconnected, Idea will log the program off, unless it includes an ON DISCONNECT statement. This statement will send program execution to the routine labeled tag; it stays there until it encounters a RETURN statement or until the program times out. Then, the monitor logs the program off.

Place the ON DISCONNECT statement with the nonexecutable statements, but not within a PROCESS statement block.

ON END DATA

[ON] END *[OF]* DATA tag

ON END DATA causes the program to branch to **tag** when the operator strikes the END DATA function key. This statement also nullifies normal operation of the END DATA key; it places the key under program control.

Place the ON END DATA statement with the nonexecutable statements, but not within the PROCESS statement block.

ON ESCAPE

[ON] ESCAPE tag

ON ESCAPE causes the program to branch to **tag** when the operator strikes the ESC key. If you don't have an ON ESCAPE statement in the program, the ESC key has the same effect as the ENTER key.

Place the ON ESCAPE statement with the nonexecutable statements, but not among the PROCESS statements.

The ESC key only has an effect if the operator is entering a value at an EDIT field.

ON FUNCTION

[ON] FUNCTION tag

This statement passes control to **tag** when the operator strikes any of four function keys, located on 6053 video terminal, while at an operator-entry field. It is nonexecutable.

The function keys are defined only for a 6053 terminal; they are the two right-most keys on the row of eight function keys. The seventh key from the left is function key 1, the eighth key is function key 2, SHIFT plus the seventh key is function key 3, and SHIFT plus the eighth key is function key 4.

The function keys act as delimiters and cause immediate exit from the field when struck. In the absence of an ON FUNCTION statement, they have the affect of a NEW LINE.

The reserved word FUNCTION allows your program to differentiate between the keys. When you strike a function key, its number is placed in FUNCTION and control passes to your program. It is up to the routine at **tag** to distinguish between the various function keys.

The value thus placed into the reserved word FUNCTION will persist until a function key is again struck.

You should define FUNCTION as a numeric register or as a field of one byte.

For example:

```
        REGISTER FUNCTION 9(1)
        ON FUNCTION ACT
              .
              .

ACT:  GO TO END,HOOK,RETRY,CHANGE, USING FUNCTION
```

ON-IOERR tag

ON-IOERR checks the setting of the file status flag, which reflects the outcome of the most recently executed I/O statement. If the flag is set (meaning that the I/O statement failed), the program branches to tag.

The system will not return serious file errors to the program. It will instead log them on the supervisory console, display a message on the associated operator's terminal advising the operator of the error, and log the operator off. Idea sends only recoverable errors to the program.

The system writes one of the following recoverable error codes into the reserved word IOERR.

Recoverable Error Codes

Code	Meaning
00	No error.
10	End of File/Subindex. The last record in the file or subindex was read by a FIND NEXT or FIND PREVIOUS statement.
18	Record Length Exceeds Block Size.
22	Duplicate Key. The key used in a FILE-NEW statement duplicates an existing key and duplicates are not allowed since no DUPLICATES COUNTED statement was specified.
23	Key is defined in the database but no record is associated with it.
24	Key doesn't exist. The key specified in a FIND USING, FIND NEAREST, FIND BEGINNING, DESTROY, REMOVE, VERIFY, or REINSTATE doesn't exist.
26	Delete denied while other pointers to record exist.
30	Physical Tape Error (such as parity).
34	End of Volume. All volumes have been exhausted.
94	Record locked. The record specified was locked by some other program. The record cannot be accessed until it is unlocked.
96	Record deleted. The record specified was logically deleted.

ON LINE-ERR

[ON] LINE-ERR tag

This statement causes the monitor to pass control to **tag** when it senses excessive (i.e., more than 64) line errors on a user's line.

When line errors are excessive and your program contains no ON LINE-ERR statement, Idea will log the console off. ON LINE-ERR allows the log-off process to be orderly. The program given control under this clause will maintain control until it RETURNs or is timed out. The next time control returns to the monitor, it will log the console off.

When Idea detects a line error and the number of line errors is not excessive, the monitor will send a message to the console operator. This message will indicate the problem and request that the user re-enter the character in question. The monitor will display the faulty character as a question mark and move the cursor to its position.

This statement is nonexecutable.

ON LOGOFF

[ON] LOGOFF tag

To log off, operators strike the LOG ON-OFF key, which initiates a normal log-off procedure. Instead, by including the ON LOGOFF statement, you can have the program branch to a routine named by **tag** when the operator strikes LOG ON-OFF.

Place the ON LOGOFF statement with the nonexecutable statements, but not within the PROCESS statement block.

If you include no ON LOGOFF statement in your program, the monitor will initiate the normal log-off sequence when the operator strikes the LOG ON-OFF key.

ON MODE CHANGE

[ON] MODE CHANGE tag

This statement branches to the routine labeled by **tag** when an operator strikes the CHANGE MODE function key. The CHANGE MODE key allows the operator to exit from a scroll area.

ON NO-ACTIVITY

[ON] NO-ACTIVITY tag

This statement passes control to the routine designated by **tag** when the specified inactivity time has elapsed. It is up to the program to then take appropriate action.

The inactivity clock is reset to zero when the program reaches each field that requires operator input. Inactivity time is the time that elapses between initiation of a field for input, and entry of the field delimiter (NEW LINE, etc.) by the operator.

The program in Figure 7-14 will log off an inactive terminal after waiting 10 minutes for operator input.

```
                NAME COFFEETIME
                INACTIVITY CONSTANT IS 10
                ON NO-ACTIVITY LOGOFF
                .
                .
                .
                PROCESS FILLER AT NONE AND BUSY
                .
                .
                .

BUSY:

                RETURN

LOGOFF:
                MESSAGE LOGGED OFF BECAUSE OF
OPERATOR INACTIVITY
                QUIT

                .
                .
```

Figure 7-14. Logging Off an Inactive Terminal with ON NO-ACTIVITY

ON-OVERFLOW

ON-OVERFLOW tag

If your program performs an arithmetic function that overflows the integer portion of its result variable, the monitor sets the overflow flag on. The ON-OVERFLOW statement checks this flag and branches to the routine labeled by **tag** if the flag is set.

For example:

```
                MULTIPLY VAR1 VAR2 RESULT
                ON-OVERFLOW MAKENOTE
                    .
                    .
                    .
MAKENOTE:       MESSAGE RESULT VARIABLE OVERFLOWED
QUIT
```

If the above multiplication resulted in a product of 8456.81 and the variable RESULT had a picture 999.99, the program would branch to MAKENOTE.

ON REPEAT

[ON] REPEAT tag

This statement passes control to **tag** when the operator strikes the REPEAT PAGE key. It is nonexecutable.

ON SCREEN

[ON] SCREEN *[IMAGE]* tag

If you use this statement in a program, the operators must be using 6053 terminals equipped with printing boards, as well as a DASHER printer. We describe this configuration in Chapter 9.

If you have the 6053 printer option, the ON SCREEN statement sends program execution to the routine labeled by tag when the operator strikes the PRINT key. The routine must contain a DISPLAY or MESSAGE statement with control codes. To print all information on the screen, use the code sequence <10> <21>. To print only the variable screen data, use <10> <01>.

This program fragment will print a snapshot of the screen when the operator strikes the PRINT key.

```
                ON SCREEN SNAPSHOT

                REGISTER FIELD 99

                      .
                      .
                      .
SNAPSHOT:       MESSAGE <10><21>
                RETURN USING FIELD
```

PARAMETERS FOR SUBINDEX

PARAMETERS *[FOR]* subindexname

This statement begins a subindex definition block. Use it and the DEFINE SUBINDEX statement to define parameters, other than the defaults, for a subindex.

The subindex definition statements are

NODE SIZE *[IS]* value

PARTIAL LENGTH *[IS]* value

The default node size is 2042, and the default partial length is 0. To determine the proper parameters for subindexes, refer to the *INFOS System User's Manual (AOS),* 093-000152.

PARTIAL LENGTH

PARTIAL LENGTH *[IS]* value

This statement specifies the partial record length associated with the subindex.

The value is the number of bytes. The default partial length is 0.

PASS

PASS recordname

Use PASS to send a record into the system COMMON file so that you can retrieve the record with another IFPL program. The other IFPL program uses an ACCEPT statement to read the record from the COMMON file.

Data that the PASS statement writes to the COMMON file will remain there until you execute another PASS statement that overwrites it.

PERFORM

PERFORM subroutinename

Use this statement to jump to a subroutine. After the monitor executes the subroutine, program control returns to the statement following the PERFORM statement.

PRINT

PRINT *[THE]* recordname USING printformatname

Use this statement to write a printing record to the system COMMON file. You define **recordname** in a **RECORD FOR PRINTING IS recordname** statement. Define **printformatname** in an **INITIATE PRINTING USING printformatname** statement, which starts a group of printing records. The program must execute the RECORD and INITIATE statements before it executes the PRINT statement.

You mark the end of the print file with a **TERMINATE PRINTING USING printformatname** statement. The program must execute this statement after it stores all the PRINT statements associated with the print format.

Figure 7-15 shows a program fragment that demonstrates how these statements fit together. FOUT is the name of the print format. To create print formats, use IFMT (or WIFMT); give the P response to the prompt TYPE(H OR P OR NONE).

```
                        SCREEN FORMAT

   NAME                  ADDRESS
   XXXXXXXXXX            XXXXXXXXXXXXXXXXXXX
   INVOICE               AMOUNT
   @  999                $999.99
   @  x


Program Segment
              RECORD FOR PRINTING IS IMAGE 1
                LENGTH IS 30
                INCLUDES NAME 1 10 ASCII
                INCLUDES ADDR 11 20 ASCII
                STOP
              RECORD FOR PRINTING IS IMAGE2
                LENGTH IS 10
                INCLUDES INV 1 3 ASCII
                INCLUDES AMOUNT 4 7 ASCII
                STOP
              RECORD FOR PRINTING IS ENDSCROLL
                LENGTH IS 1
                INCLUDES "@" 1 1 ASCII
              PROCESS NAME AT NONE AND ENAME
              PROCESS ADDR AT NONE AND EADDRESS
              PROCESS INV AT NONE AND EINV
              PROCESS AMOUNT AT NONE AND EAMOUNT
              PROCESS DONE AT FILLER AND EDONE
ENAME:        INITIATE PRINTING USING FOUT *FOUT IS
                                           *PRINT FORMAT
                                           *NAME
              STORE NAME
              RETURN
EADDR:        STORE ADDR
              PRINT IMAGE 1 USING FOUT
              RETURN
EINV:         STORE INV
              RETURN
EAMOUNT:      STORE AMOUNT
              PRINT IMAGE2 USING FOUT
              RETURN
EDONE:        PRINT ENDSCROLL USING FOUT
              TERMINATE PRINTING USING FOUT
              RETURN
```

Figure 7-15. The Statements for Printing

Because of the special nature of scroll groups, we must signal the end of the scroll group by printing an @ sign; this requires its own RECORD FOR PRINTING statement.

PRIORITY

PRIORITY *[IS]* value

Use this statement to assign a relative priority to the execution of the local Idea process under which your application runs. The legal values are 1, 2, and 3, where 1 is the highest priority.

Each user profile in AOS has a priority established with the Profile Editor (PREDITOR). The PRIORITY statement can lower this priority, but not raise it.

PROCESS

[label#] PROCESS { FILLER / variable } *[AT]* { NONE *[AND]* tag_2 / tag_1 *[AND]* NONE }

PROCESS statements regulate the main flow of control in Idea programs by sending program control to screen-field-related routines. Also, together with the screen-format field pictures, they declare program variables.

In the format above, **variable** is the name of a screen field.

If you give the field the EDIT attribute (but not DISPLAY) with IFMT, use a PROCESS statement of the form

PROCESS variable AT NONE and tag_2

where tag_2 labels a routine that will process the EDIT-field variable; this routine must contain a STORE statement.

If you give the field the DISPLAY attribute (but not EDIT), use a PROCESS statement of the form

PROCESS variable AT tag_1 AND NONE

where tag_1 labels a routine that will process the DISPLAY-field variable; this routine must contain a DISPLAY statement.

If you give both the EDIT and DISPLAY attributes to a screen field, use this form of the PROCESS statement:

PROCESS variable AT tag_1 AND tag_2

where tag_1 labels a routine that will use the field as a DISPLAY field, and tag_2 labels a routine that will use the field as an EDIT field.

FILLER is a reserved word that lets you use variables declared with REGISTER statements or variables that will never receive a value. We explain this in detail below.

The optional *label* lets you direct program execution to the PROCESS statement with a RESET or a RETURN label statement. You must place a pound sign immediately after the label, with no spaces in between, such as PAYDAY#. You must place a space or a tab after the pound sign, as in this PROCESS statement:

```
PAYDAY#  PROCESS WAGES AT NONE AND PAYCHECK
```

The statement RETURN PAYDAY will direct program execution to this PROCESS statement, which directs execution to the routine PAYCHECK to process the variable WAGES.

IFPL programs must contain a PROCESS statement for each logical field in the associated format. (A logical field has the EDIT and/or the DISPLAY attribute.) The monitor orders the screen fields from left to right and from top to bottom. The PROCESS statements must follow this order; i.e., the first PROCESS statement must correspond to the first logical screen field, the second PROCESS statement to the second logical screen field, and so on.

Using the reserved word FILLER in a PROCESS statement in place of **variable** can save you space in certain instances. Use PROCESS FILLER to display program constants, or in places where you do not have to allocate space for the variable.

The following program fragment demonstrates the use of REGISTER, PROCESS FILLER, and the REGISTER variable within a routine.

```
        PROCESS FILLER AT E1 AND NONE

        REGISTER AA X(11) CORPORATION

E1:     DISPLAY AA
        RETURN
```

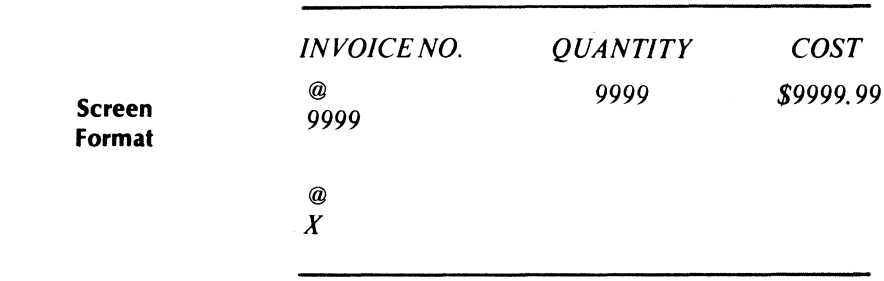
In Figure 7-16, we use a dummy field to send program execution to a routine after an operator has completed a scroll area. The field has no other use, so we don't need to allocate space for it.

In Figure 7-16, we want to pass data from one program to another and then Link to the second program. We want these things to occur after an operator has completed the scroll area entries. We don't know ahead of time how many entries the operator will make, and we cannot place the PASS and LINK statements in the E1 routine. (We want to send the data *after* we've finished the scroll area.) So, using IFMT, we place a dummy field on the screen format (the single X), and give it the DISPLAY attribute so the monitor will pass control directly to the program without waiting for operator input. In the program, we use the PROCESS FILLER statement to direct program execution to the routine D1, which performs the PASS and LINK tasks.

Notice that routine D1 doesn't display any data on the screen.

You may use as many PROCESS FILLER statements in one program as you need, as long as you maintain the proper correspondence with the screen fields. However, the word FILLER is meaningless in any other IFPL statement.

Screen Format



IFPL Program

```
PROCESS COST AT NONE AND E1
PROCESS FILLER AT D1 AND NONE
.
.
.
E1: STORE COST
    RETURN
D1: PASS PARAMETERS
    LINK USING FORMAT2
    FINISH
```

Figure 7-16. An Example of PROCESS FILLER

QUEUE

QUEUE variable

The QUEUE statement lets you queue a batch job from an IFPL program.

The **variable** is any type of IFPL variable, including a literal. It must be alphabetic or alphanumeric; it cannot be numeric.

Also, **variable** can be any CLI command or macro. For example, we will explain what happens when a program executes the following QUEUE statement:

```
QUEUE "QPRINT MY_FILE"
```

Idea first creates a batch job file. It then places the contents of the variable -- which in this case is the literal QPRINT MY_FILE -- in the batch file. It then queues the batch file. When the batch stream executes this job, it will execute the CLI command QPRINT MY_FILE.

You can give a series of commands in a QUEUE statement variable by separating the commands with semicolons (;),

```
QUEUE "DIR :UDD:JTM:IDEABOOK;SYNTAX/L CHECKBOOK:QPRINT CHECKBOOK.UP"
```

or you can create a macro that contains a series of commands, and then give the macro's name as the contents of **variable**.

The system uses the initial working directory and search list for QUEUE commands. Therefore, be sure that any files that QUEUE commands will need are within that directory, or are in directories appearing in the initial search list. Also, remember that the system places any files created by the QUEUE CLI commands within that directory.

QUIT

QUIT

The QUIT statement terminates a program. When the monitor executes this statement, it closes any open files and logs the operator off. The QUIT statement does not clear the terminal screen.

RANGE

RANGE variable$_1$ variable$_2$ variable$_3$

RANGE compares the contents of the three specified variables to determine whether the contents of variable$_2$ lie within the limits of variable$_1$ and variable$_3$. If variable$_2$ is greater than or equal to variable$_3$, Idea sets the IN-RANGE flag; otherwise is sets the OUT-RANGE flag.

RANGE evaluates fields in a fashion similar to COMPARE; it compares numeric fields numerically (only), and alphanumeric fields by field length and character. It treats dissimilar fields as alphanumeric with a SYNTAX warning message. Typical RANGE operations are shown in Table 7-6.

The statements IN-RANGE and OUT-RANGE provide conditional branching depending on the results of a RANGE statement.

Table 7-6. Typical Operations

variable$_1$	variable$_2$	variable$_3$	Flag Set
17	017.0	23.6	IN-RANGE
7	0.7	8.10	OUT-RANGE
AAF	DLM	XYZ	IN-RANGE
SPECNO	PTNO	IDNO	OUT-RANGE

RECEIVE

RECEIVE recordname *[[FROM] ipc-port-name]*

This statement gets a record from the specified IPC port. The default IPC port is the RCX70 port. To use the RCX70 port, you must have specified this option during the IDEASG dialog (see Chapter 10).

The *ipc-port-name* can be a literal or a register that contains the name of an existing port.

To receive a record from another Idea console, you attach that console's number to the keyword IDEA. For example, to receive a record sent to your Idea console from Idea console 4, you would use this statement:

```
RECEIVE ACCTREC FROM "IDEA04"
```

The program running on Idea console 4 would give a SEND command, such as:

```
SEND ACCTREC TO "IDEA07"
```

You can also receive a record from a non-Idea process, such as a COBOL program. However, the non-Idea process must create a port and give it a name before you can use this name in your program's RECEIVE statement. You must therefore know what this name is.

You may receive the following errors in the IFPL register IOERR:

Error Code	Explanation
54	No outstanding message to be received.
55	SEND error.
56	RECEIVE error.

If there is no message to be received, your program will not wait for one, but will signal an error, placing the code 54 in IOERR.

You must execute a RECEIVE statement immediately after executing a SEND REQUEST statement (see the SEND statement).

There are several conditions that can cause errors 55 and 56. They can range from a nonexistent IPC port to the ACL of the port not allowing you access. To find out the exact error, examine the AOS error code in the INFOS-ERR register. INFOS-ERR may also receive RCX70 errors; refer to the *RCX70 Reference Manual (AOS).*

RECORD

RECORD *[FOR]* {filename / subindexname} *[IS]* recordname

This statement associates **recordname** with **filename** or **subindexname**. It also starts the record definition block, which defines the lengths and layouts of fields within the record. You can have three types of statements within a record definition block: LENGTH, INCLUDES, and REDEFINES. You must terminate every record definition block with a STOP statement. For example, the following are typical record definitions.

```
RECORD FOR STAFF_VIEW IS OFFICE
LENGTH IS 8
INCLUDES OFFICE_NO 1 3 ASCII
INCLUDES TEL_EXT 4 5 ASCII
STOP

RECORD FOR STAFF_VIEW IS FOO
REDEFINES OFFICE
STOP
```

RECORD FOR PASSING

RECORD *[FOR]* PASSING *[IS]* recordname

RECORD FOR PASSING defines a record that you may use with a PASS or ACCEPT statement. It is a special form of the RECORD statement, but it conforms to the format for record descriptions. (See RECORD.) The word PASSING differentiates between a passing record and a normal database record.

You need no FILE statement with RECORD FOR PASSING. The remainder of the record definition block is the same as for any other type of record; that is, use the LENGTH statement and INCLUDES statements (terminated by STOP) to indicate the fields that will make up the record.

Use RECORD FOR PASSING, PASS, and ACCEPT to pass the value of variables between two or more programs running on a single terminal.

If you need to link to another IFPL program via a LINK statement, you may need to transfer information between the two programs. To accomplish this, use a PASS statement to transfer a record to the system COMMON file and an ACCEPT statement to accept all or part of the previously passed data into the current program for processing. As stated above, you must use a RECORD FOR PASSING statement to define the record being PASSed.

You cannot use **recordname** assigned in a RECORD FOR PASSING statement in any context other than a PASS or ACCEPT statement.

RECORD FOR PRINTING

RECORD *[FOR]* PRINTING *[IS]* recordname

This statement defines a record that you will use to write printing records to the COMMON file for printing with the PRINTF utility. Record definitions for printing are identical to other record definitions.

RECORD FOR TAPE

RECORD *[FOR]* TAPE *[IS]* recordname

RECORD FOR TAPE is another special form of the RECORD statement; use it to define records for logging to magnetic tape.

The information you give in the record description block that follows a RECORD FOR TAPE statement is all that your IFPL program requires for logging records to tape.

REDEFINES

REDEFINES recordname

This statement allows you to use a record definition block that you created for one record for other records. The redefined records can be in the same file as the original or in other files. This saves you time when typing programs.

You do not use a LENGTH statement or INCLUDES statements with REDEFINES. You must, however, place a STOP statement after each REDEFINES statement.

For example:

```
RECORD FOR FILE1 IS AREC
        LENGTH IS ...
        INCLUDES ...

STOP

RECORD FOR FILE2 IS BREC
REDEFINES AREC
STOP

RECORD FOR FILE1 IS CREC
REDEFINES AREC
STOP
```

Note that you cannot redefine a record that you declare with a REDEFINES statement. For example, you could not redefine BREC or CREC onto another record; you would have to use AREC again.

To use an INVERT statement, you must use a REDEFINES statement. (See INVERT.)

REDESIGNATE

REDESIGNATE register

REDESIGNATE defines a portion (or portions) of a **register** so that you may reference it (or them) separately. You may use only one REDESIGNATE statement per register, and it must immediately follow the register to which it refers. There is no limit to the number of subregisters defined for a register and they may overlap in any way. For example, see Figure 7-17.

You may use the redesignated fields in any context suitable for a register. Each subregister requires two arguments: the first must reference the starting character or byte in the register, and the second must indicate the length of that subregister. A STOP statement must follow the last subregister definition.

```
REGISTER DATE X(8) 00/00/00
REDESIGNATE DATE
MMONTH 1 2          *STARTS AT BYTE 1 OF DATE, 2
                    *BYTES LONG
MDAY 4 2            *STARTS AT BYTE 4 OF DATE, 2
                    *BYTES LONG
MYEAR 7 2           *STARTS AT BYTE 7 OF DATE, 2
                    *BYTES LONG
MONTH/DAY 1 5       *STARTS AT BYTE 1 OF DATE, 5
                    *BYTES LONG
STOP
```

Figure 7-17. Use of REDESIGNATE

REFILE

REFILE *[THE]* recordname USING key...

This statement updates a record. Therefore, the keys should be the same as those used in the FIND statement to locate the record.

Also, REFILE automatically releases a locked record when it refiles the record in the database.

REGISTER variable picture *[initial-value]*

The REGISTER statement allows you to create internal variables or constants. There is no limit to the number of registers that you may define, but each **variable** must be unique.

The **picture** field must consist of as many Xs (alphanumerics), As (alphabetics), or 9s (numerics) as are necessary to define the length of the field. A numeric field may contain a decimal point and/or a sign indicator (use the letter S) at either end of the field. You may also define a picture by declaring a character count in parentheses after you declare the string type as A, X, or 9; thus **X(6)** is equivalent to **XXXXXX**.

The *initial-value* must conform to the data type specified by the picture. If you give no *initial-value,* the field will be initialized to blanks (for pictures specified by As or Xs) or 0s (for pictures specified by 9s). For example:

```
REGISTER A XXXXXX FALSE
```

assigns the constant FALSE to register A, whereas:

```
REGISTER ZERO S9(4).9(2)
```

assigns the value +0000.00 to a register named ZERO.

The REGISTER statement stores the sign of a signed field on the left or right of the signed value, depending on where you place the sign designator in the picture.

If you will use the register in a DISPLAY or STORE statement, you must describe the screen field exactly as you defined the register picture. For example, if your REGISTER statement places the sign on the right, so must your screen picture. Using PROCESS statements instead of REGISTER statements removes this concern.

You may place characters not included in the standard set of alphanumerics (that is, ASCII characters outside the range 40_8 to 176_8, inclusive) in REGISTER statements. Simply enclose the code of each such control character in angle brackets; for example, list ASCII code 7 as <7>, <07>, or <007> (the system is tolerant of leading zeros).

There is no limit to the number of codes that you can use in a string. The code for a character, plus the angle brackets that enclose it, are equivalent to one character of data, and any preceding or trailing blanks are counted in the string. For example, the statement

```
REGISTER DATA X(8) <7> EXAMPLE
```

assigns an initial value of <7>EXAMPLE to the register named DATA. The register is eight characters long. It includes the seven letters in the word EXAMPLE preceded by the single ASCII character 007.

REINSTATE

REINSTATE *[THE]* recordname USING key...

This statement reinstates a logically deleted record. The REMOVE statement logically deletes records (see REMOVE).

RELEASE

RELEASE *[THE]* recordname USING key...

This statement frees the record **recordname** for access by other programs, but retains the record for use by your program.

RELEASE ALL

RELEASE ALL HOLDS *[IN]* filename

This statement unlocks all records locked by the current program. The current file position is not changed.

REMOVE

REMOVE *[THE]* recordname

This statement logically deletes a record; the record exists, but the system erases the pathway to it. To rebuild the pathway to the record, use the REINSTATE statement.

If you attempt to access a logically deleted record, IOERR is set to 96 (Record Logically Deleted).

RESET

RESET { field number / label }

RESET field number

will reset logical field number **n** to DISPLAY mode. (See RESET USING.)

RESET label

will reset the field identified by label to DISPLAY mode, where **label** indicates a labeled PROCESS statement. (See RESET USING.)

You can use only one RESET statement per field-processing routine. If you use more than one, only the last one will have an effect.

RESET doesn't reset the field until control passes back to the monitor.

RESET USING

RESET USING variable

This statement tells the monitor to reset to DISPLAY the field whose number is contained in **variable.** It has meaning only for fields defined as both DISPLAY and EDIT.

The monitor resets the field to DISPLAY when it regains control; that is, following the execution of a RETURN statement. The DISPLAY attribute will take effect the next time that the monitor processes the designated field.

You can reset only one field at a time. To reset a second field, you must re-enter the program.

Note that RESET's execution does not imply a RETURN.

For example:

```
      PROCESS F1 at D1 and E1

D1:   DISPLAY ABC
      RETURN USING FIELD
E1:   STORE F1
      RESET USING FIELD * FIELD WILL BE "DISPLAY" NEXT TIME
      RETURN
```

RETRIEVE HIGH KEY

RETRIEVE HIGH KEY *[FOR]* recordname *[TO]* variable

This statement retrieves the highest key for **recordname** at the current INFOS system level. "Highest" means the key with the highest ASCII value; for example, ZZZ is higher than AAA, and AAA is higher than 253. The current INFOS system level means that before you use the RETRIEVE HIGH KEY statement, you must use a FIND or a VERIFY statement to position to the index level you want.

RETRIEVE HIGH KEY places the retrieved key value in **variable**. You can then use **variable** in file manipulation statements.

Figure 7-18 illustrates this statement.

To postition to the proper subindex, we use:

```
VERIFY THE INVOICEREC USING "SMITH", "01"
```

To retrieve the highest key in the subindex, we then use

```
RETRIEVE HIGH KEY FOR INVOICEREC TO KEYNUMBER
```

KEYNUMBER will then contain 77.

If we had specified duplicates with a DUPLICATES statement, the system would place the duplicates count in the variable we specified in the DUPLICATES statement.

You cannot use RETRIEVE HIGH KEY before using a FIND or VERIFY statement; this will result in a fatal INFOS system error.

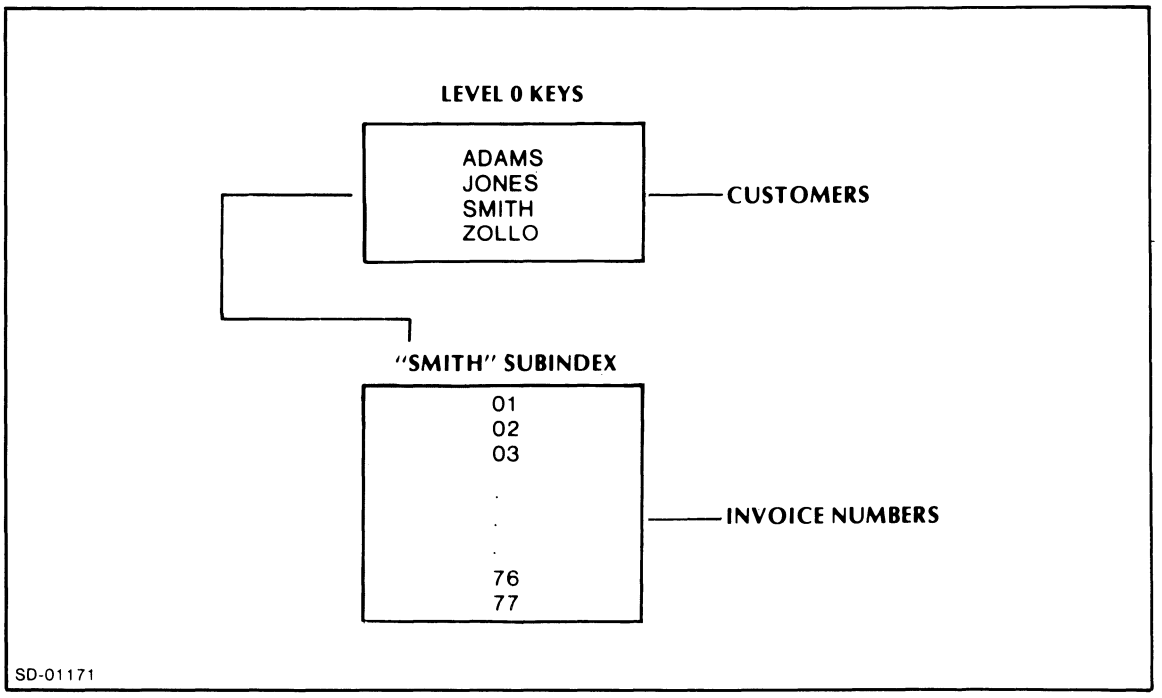


Figure 7-18. Retrieving the Highest Key

RETRIEVE KEY

RETRIEVE KEY *[FOR]* recordname *[TO]* variable

Use this statement with the FIND BEGINNING and FIND NEXT statements; these do not return a key or a duplicates count.

RETRIEVE KEY places into **variable** the last key that was entered in a FIND statement key string. For example:

```
FIND THE INVOICEREC USING "SMITH","43"

RETRIEVE KEY FOR INVOICEREC TO CURRENTKEY
```

CURRENTKEY will contain the value 43.

The system performs any necessary conversions from BINARY or PACKED to ASCII. Also, if it finds duplicates at the level of the retrieved key, it enters the duplicates count in the variable specified by the DUPLICATES statement.

You may use the reserved word VARIED-KEY for **variable** if the keys you want to retrieve have different lengths.

Figure 7-19 demonstrates the use of RETRIEVE KEY in a program that updates a database by deleting unnecessary records. Since the program uses FIND BEGINNING and FIND NEXT statements, we need the RETRIEVE KEY statement to keep track of the record keys so that we can use the REMOVE statement on the records.

```
NAME UPDATE

FILE CLIENTFILE
        KEY FOR CLIENTFILE IS 10 ASCII
        SUBINDEX FOR CLIENTFILE IS NAMEKEY
        KEY FOR NAMEKEY IS 30 ASCII
        DUPLICATES ARE COUNTED IN DUPECOUNT
        RECORD FOR NAMEKEY IS CLIENTREC
                LENGTH IS 200
                INCLUDES NAME 1 30 ASCII
                INCLUDES ADDRESS 31 30 ASCII
        STOP

PROCESS AREAKEY AT NONE AND E1
PROCESS PARTIALKEY AT NONE AND E2
PROCESS NAME AT D3 AND NONE
PROCESS ADDRESS AT D4 AND NONE
PROCESS CHOICE AT NONE AND E5

REGISTER DUPECOUNT 9(4)
REGISTER KEYFOUND X(30)

ON ESCAPE ESC

*OPERATOR IS LOOKING FOR SMITH, STANLEY J., 132 WEST 57TH STREET
*OPERATOR ENTRY IS EAST IN FIELD 1, SM IN FIELD 2

E1:
        STORE AREAKEY
RETURN
```

Figure 7-19. Name Update

```
E2:
        STORE PARTIALKEY
        FIND THE CLIENTREC BEGINNING WITH AREAKEY,PARTIALKEY
        ON-IOERR TRYNEWKEY
RETURN


*THE NEXT FIELD IS THE START OF A SCROLL AREA
*THE CLIENT NAME AND ADDRESS IS DISPLAYED
*THEN THE OPERATOR HAS THE OPTION OF DELETING THE RECORD
*IF THE RIGHT ONE HAS BEEN RETRIEVED, OR OF CONTINUING
*TO SEARCH THE FILE


D3:
        DISPLAY NAME
RETURN


D4:
        DISPLAY ADDRESS
RETURN


E5:
        COMPARE CHOICE "Y" *Y = DELETE
        IF EQUAL E5A *GO TO DELETE ROUTINE
        FIND THE NEXT CLIENTREC
        ON-IOERR TRYNEWKEY
RETURN          *TO BEGINNING OF SCROLL AREA


*CONTROL PASSES TO THE NEXT ROUTINE ONLY IF THE OPERATOR HAS
*KEYED IN THE LETTER "Y" IN FIELD 5, SIGNALLING THE
*DELETION OF THE RECORD DISPLAYED IN THE SCROLL AREA

E5A:
        RETRIEVE KEY FOR CLIENTREC TO KEYFOUND

*KEY IS NOW IN KEYFOUND, DUPLICATES COUNT IN DUPECOUNT
*THE LENGTH OF THE KEY TO BE RETRIEVED MUST BE THE
*SAME AS THAT SPECIFIED FOR THE DESTINATION
*VARIABLE  (KEYFOUND)

        REMOVE THE CLIENTREC USING AREKEY, KEYFOUND
        ON-IOERR E5B
        MESSAGE RECORD LOGICALLY DELETED. ENTER NEW KEY TO CONTINUE
RETURN 1


E5B:
        MESSAGE UNABLE TO DELETE CURRENT RECORD
RETURN 1

TRYNEWKEY:
        MESSAGE NO MORE CLIENTS WITH CURRENT KEYS
RETURN 1

ESC:
RETURN 1

*ESCAPE ALLOWS OPERATOR TO REFINE  THE PARTIAL KEY WITHOUT
*HAVING TO SEARCH THE ENTIRE FILE

FINISH
```

Figure 7-19. Name Update (continued)

RESTART

RESTART

RESTART returns the cursor to the first field on the screen, resets the DISPLAY/EDIT flip-flop of the DISPLAY and EDIT fields to DISPLAY, and erases unprotected data from the screen. It does not reinitialize program variables.

RETURN

RETURN { *[field-number]* / *[label]* }

The RETURN statement is the normal statement you use to return process control to the Idea monitor so that it can determine the next field to process. Used without *field-number* or *label,* it returns control to the next PROCESS statement. If there isn't another PROCESS statement, control passes to the FINISH statement.

RETURN field-number

returns control from a routine to the specified field's PROCESS statement. For example, RETURN 3 passes control to the PROCESS statement of the third screen field.

RETURN label

returns control from a routine to the PROCESS statement specified by **label**. This label must be a PROCESS statement label, not a tag.

For example, the statement

A1 # PROCESS F1 AT NONE AND E1

shows a PROCESS statement with the label A1. The pound sign (#) is the label delimiter.

RETURN USING

RETURN USING variable

This statement returns to the physical field corresponding to the value of **variable**. If **variable** contains a value which is less than 1 or greater than the number of fields in the format, the system will ignore the argument and return to the next field.

If you use the reserved word FIELD for **variable** (RETURN USING FIELD), the system will return to the field currently being processed.

RIGHT

RIGHT *[JUSTIFY]* variable$_1$ *[IN]* variable$_2$

This statement will right justify a smaller source field in a larger destination field. It moves data from variable$_1$ to variable$_2$ starting with the right-most character position and proceeding from right to left.

The RIGHT statement treats blanks like any other character. It will perform no zero- or blank-filling in the destination field. If the destination field is longer than the source, the system will retain the excess destination data.

The system will disregard a decimal point in a source field. It will display a decimal point in a destination field as it is specified in the field picture.

The system performs transfers of similar data types between fields of equal size on a character-position-by-character-position basis. No justification is involved in such moves since the system treats blanks as data.

Table 7-7 shows some examples of data moved using the RIGHT command. In this table, Dest means Destination and Srce means Source, which remains unchanged.

Table 7-7. Examples of Data Moved with the RIGHT Command

Example	Initial Values	Final Dest Values
Numeric Srce < Dest No Decimal Point	Srce = 788 Dest = 55555	55788
Numeric Srce > Dest No Decimal Point	Srce = 83492 Dest = 671	492
Numeric Srce > Dest No Decimal Point	Srce = 16.98 Dest = 178.544	171.698
Numeric Srce > Dest Decimal Point	Srce = 856.99 Dest = 78.5	69.9
Alphanumeric Srce = Dest	Srce = patnum Dest = vacancy	vpatnum
Alphanumeric Srce > Dest	Srce = patnum Dest = vac	num
Mixed Srce < Dest	Srce = 858.9 Dest = station	sta8589
Mixed Srce > Dest	Srce = sub Dest = 6.3	ub

SEND

SEND { recordname *[[TO] ipc-port-name]* / REQUEST recordname }

SEND recordname *[[TO] ipc-port-name]*

sends a record to an IPC port. The *ipc-port-name* must be the name of an existing IPC port. You can use literals and registers for your port names.

If you are using the RCX70 port, you do not give the phrase

TO ipc-port-name

To use RCX70, you must attach the /IPC switch to the IDEASG command (see Chapter 10).

To send a record from your Idea console to another Idea console, you attach the number of the receiving console to the keyword IDEA. For example, to send a record to console 7, you would give this command:

```
SEND ACCTREC TO "IDEA07"
```

The program running on console 7 has to issue a RECEIVE statement in order to receive the record:

```
RECEIVE ACCTREC FROM "IDEA04"
```

You can also send a record to a non-Idea process such as a COBOL program. The non-Idea process must create a port and give it a name. You then use this name in the SEND statement.

The statement

```
SEND REQUEST recordname
```

is only valid for RCX70 applications. It tells RCX70 that you want to receive a message from the host machine. (With RCX70, the **SEND recordname** statement means that you do not expect a reply.) The contents of **recordname** may be a null (dummy) record, a key for the remote database, a record for the remote database, or any other convention that the host and local applications decide upon.

The program must issue a RECEIVE statement immediately after it issues the SEND REQUEST statement, so that it will be ready when the host responds. You should loop on the RECEIVE statement until you receive the message. Use IOERR error code 54 (RECEIVE error -- no message ready) to loop.

In normal cases, you will receive the record that you requested. If the host does not respond during the time-out period, you will receive a time-out error message.

You may receive the following error codes in IOERR with any form of the SEND statement:

IOERR Error Code	Explanation
54	RECEIVE error -- no message ready
55	SEND error
56	RECEIVE error

There are several conditions that can cause errors 55 and 56. They range from a named but nonexistent IPC port to an incorrect ACL for the port. The register INFOS-ERR will contain the actual AOS error code. You may also receive RCX70 error codes in INFOS-ERR -- refer to the *RCX70 Reference Manual (AOS).*

STOP

STOP ends record description blocks, register redesignations, and parameters for subindex definition blocks.

To end a record description block, place STOP immediately after the last INCLUDES statement. STOP must also follow every REDEFINES statement.

STORE variable

STORE reads input data into memory by taking data entered into the current screen field and storing it in a register called **variable**. This is a crucial EDIT field statement.

Before you can manipulate any data entered through the keyboard, you must store it in a memory location that bears a variable name associated with a PROCESS or REGISTER statement.

Typically, STORE is the first statement in an edit routine, and **variable** represents sufficient buffer memory to store any value that the operator keys in. As a result, you usually declare **variable** with a PROCESS statement that is associated with the current field at compile time. The **variable** receives the buffer-memory storage characteristics you specify in the PROCESS-statement-related screen field.

For example:

```
PROCESS ADDRESS AT NONE AND NAME
        .
        .
        .
ENAME: STORE ADDRESS
        .
        .
        .
```

SUBINDEX

SUBINDEX *[FOR]* {filename / subindexname$_1$} *[IS]* subindexname$_2$

This statement specifies the name of the subindex. It also allows the compiler to keep track of the number of keys required to access a record defined at the given subindex level.

SUBINDEX statements must appear in order, from the lowest level to the highest level. For example, you must define subindex A of file 1 before you define subindex A2 of subindex A.

SUBROUTINE

SUBROUTINE name

SUBROUTINE must be the first statement in a subroutine. It declares the name of the subroutine. All following IFPL source statements are part of the subroutine until the ENDSUB statement appears.

SUBTRACT

SUBTRACT subtrahend minuend difference

SUBTRACT subtracts the contents of **subtrahend** from the contents of **minuend** and stores the result in **difference.**

To ensure that you don't lose valuable digits by truncation, give the **difference** one more integer place than the larger of the **minuend** and **subrahend.**

TABLE

TABLE name

Use this statement to define tables in your IFPL program. Follow this statement with a list of table elements and end with the statement ENDTABLE. Once you define a table, you can use other statements to perform table lookups and to extract table elements by index.

You may define a maximum of 40 tables within your IFPL program; however, there is no limit to the number of elements within a table.

The table elements may be any mix of register names, PROCESS variables, or literals. The elements can have different lengths.

When you access table elements, literals return exactly as you entered them in the table. Registers and variables return their contents; i.e., the value stored in the register or variable location.

If your program uses a literal from a table and it changes the literal's value in some way, it will then store the new value in the table.

For example:

```
TABLE ERRORCODES
"00"
"10"
"22"
"23"
"24"
"94"
"96"
ENDTABLE
```

defines a table where all of the elements are literals.

```
TABLE MESSAGES
OKMESSAGE
ENDOFFILE
"22"
NORECORD
KEYTOOBIG
RECDLOCK
DELETED
ENDTABLE
```

defines a table where elements are a mix of literals and program variables.

Table elements should be variables or distinctive literals. You should not use dummy literals; for example,

```
TABLE DUMMYLITERAL
    " "
    " "
    " "
ENDTABLE
```

and the statement

```
MOVE "JANE" TO DUMMYLITERAL (ENTRY)
```

In this case, the MOVE statement will give the value JANE to the table element pointed to by the value of ENTRY. But, since the table element names are the same, all the other table elements will also take on the value, JANE. This would also destroy the space characters as defined by the literal "□□". Consequently, a COMPARE involving the literal "□□" would compare a value against the value JANE, not against the two space characters.

Use the LOOKUP statement to search a table. The system sets a flag to the index of the matching element if it finds the element. If it doesn't find a match, the system sets the flag to 0.

You may use the value returned in the LOOKUP pointer to extract table elements via the DISPLAY and MOVE statements. For example, suppose that you have this table:

```
TABLE SSNUMB
"020349912"
"726886990"
"012526722"
"555122223"
"909090909"
ENDTABLE
```

The statement

```
LOOKUP IN SSNUMB(MPTR) "012526722"
```

will locate the third social security number and place the value 03 in the pointer MPTR. You can then give the statement

```
DISPLAY SSNUMB(MPTR)
```

to display the social security number 012526722 in a field with the DISPLAY attribute.

TERMINATE

TERMINATE PRINTING USING printformatname

This statement marks the end of the print format in the COMMON file. It tells the PRINTF utility that the print format is complete and ready to be printed.

VERIFY

VERIFY *[THE]* recordname USING key...

This statement positions you to the record **recordname** without incurring the overhead of reading the record (as with the FIND statement).

VERIFY won't tell you whether a record is locked.

VERIFY NEXT

VERIFY *[THE]* NEXT recordname

This statement positions you to the next record, but doesn't retrieve it. Use it to skip over locked records.

VERIFY PREVIOUS

VERIFY *[THE]* PREVIOUS recordname

This statement positions you to the previous record, but doesn't retrieve it. Use it to skip over locked records.

End of Chapter

Chapter 8
Idea System Utilities

In this chapter, we describe the Idea system utility programs. Table 8-1 lists the utilities and their functions.

Table 8-1. The Idea Utilities

Utility	Function
ALPHA	Redefines the legal alphabet.
CHGEM	Builds new system dialog files.
DEFCOM	Defines the COMMON file.
ILIB	Builds a format library.
PALPH	Prints the current alphabet.
PFMT	Prints or displays information about formats.

ALPHA

Redefines the Alphabet

Use the ALPHA utility to redefine the alphabet. You may wish to do this, for example, to change the decimal point to a comma for European usage, or to change the currency symbol from the dollar sign to another symbol.

To use ALPHA, give this command from the CLI:

```
ALPHA)
```

ALPHA will display the current decimal character, currency symbol, and alphabetic characters, and will ask you if you want to change them. If you answer Y (for YES), the system asks you for the new characters. It then tells you it has created a new ALPHABET.TB file, displays the new characters, and returns to the CLI. Figure 8-1 shows the entire dialog.

```
)ALPHA)

CURRENT ALPHA DATA IS:
DECIMAL POINT IS,
CURRENCY SYMBOL IS %
ALPHABET IS
□ABCDEFGHIJLMNOPQRSTUVXYZ
CHANGE ALPHA DATA? (Y OR N)  Y
PLEASE ENTER YOUR DECIMAL POINT CHARACTER
(OR ) FOLLOWED BY A CARRIAGE RETURN.

PLEASE ENTER YOUR CURRENCY SYMBOL FOLLOWED BY A CARRIAGE RETURN.

$

CHARACTERS VALID FOR ENTRY INTO AN ALPHABETIC FIELD.
INCLUDE THE SPACE CHARACTER AND YOUR ALPHABET.
PLEASE ENTER ALL OF THESE CHARACTERS TERMINATING WITH A CARRIAGE RETURN

□ABCDEFGHIJKLMNOPQRSTUVWXYZ)

FILE ALPHABET.TB HAS BEEN BUILT.
CURRENT ALPHA DATA IS:
DECIMAL POINT IS.
CURRENCY SYMBOL IS $
ALPHABET IS

□ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Figure 8-1. A Sample ALPHA Dialog

CHGEM

Changes the Dialog Files

To change a system dialog file, you first edit the file with SPEED or LINEDIT. Then, you process the file with the CHGEM utility.

Editing the Source Files

The system messages are in source files with the extensions .AOS.ER. When editing these files, restrict your changes to the messages themselves. For example, the error-message source file GIDEA.AOS.ER contains these lines:

```
CODE  4
.TXT    /TOO MANY CHARACTERS ENTERED; RE-ENTER THE COMMAND<012>/
```

The actual message is

TOO MANY CHARACTERS ENTERED; RE-ENTER THE COMMAND 012

The message field begins with the first nonspace, nontab character following .TXT. The program uses the first character it encounters as the message delimiter; you can use any graphic character on the keyboard, except the semicolon and the angle brackets. For example, if you want to use a slash within the message itself, you can use the question mark or some other character as the delimiter, as in this message:

```
.TXT ?TOO MANY KEY/SUBINDEX DEFINITIONS?
```

You cannot use the semicolon as the delimiter because it begins comment fields. You cannot use the angle brackets as delimiters because they set off octal control codes.

Do not edit the lines containing the word CODE, nor the characters .TXT(TAB/SPACE).

Processing the Message File with CHGEM

After you have edited the message file, give this command from the CLI:

CHGEM root-error-filename *[PRINT]*)

where:

root-error-filename is the name of the error file minus the .AOS.ER extensions.

You may use angle brackets and other CLI command templates in the CHGEM command line.

The optional argument *PRINT* sends a copy of the assembled list file and the load map file to the line printer.

CHGEM uses the root-error-filename.AOS.ER file as input, and outputs the file root-error-filename.ER.

For example:

```
CHGEM DIALOG PRINT)
```

creates the file DIALOG.ER from the source file DIALOG.AOS.ER and sends a copy of the assembled file to the line printer.

DEFCOM

Defines the COMMON File

To create the system COMMON file, give this command:

```
DEFCOM)
```

This creates a standard COMMON file. We discuss the structure of COMMON and show you some ways that you can alter this basic structure in Appendix B.

The COMMON file is blank when you first create it. After you use it with print records and formats, it contains the print record information.

To delete old print records from COMMON, you run DEFCOM again. The system will tell you that the COMMON file exists and will ask if you wish to delete it. After you type a D to delete, DEFCOM creates a new blank COMMON file. You can also use a PRINTF feature that deletes records as it prints them.

ILIB

Builds a Format Library

Use the ILIB utility to build a library of formats.

To run ILIB, you must be in the same directory as the formats. To use the library, you must specify the library name during the IDEASG dialog (see Chapter 10). Linking from format to format may run faster if you use a library.

You may run a format-library local monitor and a non-format-library local monitor under the same global monitor.

To build your library, give this command:

```
XEQ ILIB libraryfilename.FPL)
```

where:

libraryfilename.FPL is the name that the system manager will specify in the IDEASG dialog. The name must have the suffix .FPL.

ILIB will then display the screen shown in Figure 8-2.

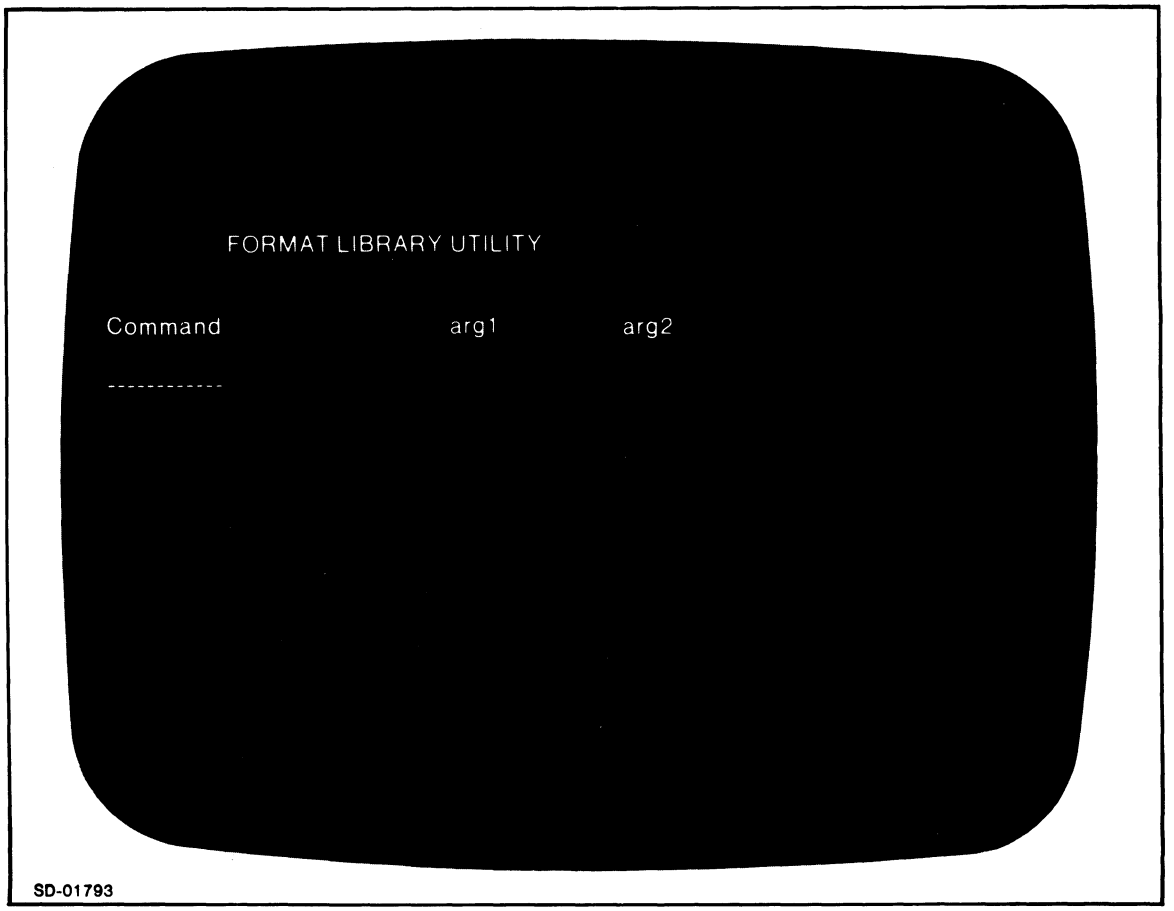


Figure 8-2. The ILIB Screen

When you type one of the commands shown in Table 8-2, ILIB will ask you to enter the arguments that are appropriate for that command.

Table 8-2. The ILIB Commands

Command	Description
BUILD	Takes all files in the directory with the suffix .FP and places them in the library.
MERGE	Searches the working directory for .FP files with the same names as those in the library. If it finds a match, it replaces the format in the library with the .FP file in the directory.
ADD	Adds a format to the library. Displays an error if the format already exists in the library or if the library is full (512 formats).
ANALYZE	Lists the formats in the library and the dates when each format was added. The default listfile is @CONSOLE; however, you can change this.
BYE	Returns you to the CLI.
DELETE	Deletes the specified format from the library.
RENAME	Renames a format in the library. You first specify the existing format, and then give the new name.
REPLACE	Replaces the named format with a format bearing the same name.

Executing in Batch Mode

You may run the BUILD or MERGE commands in batch mode. Give this command:

QBATCH XEQ ILIB libraryname.FPL {BUILD / MERGE}

Moving the Library

You must create the library in the directory where the formats reside. After you create it, you can move the library to any directory you choose.

Prints the Current Alphabet

To print the current alphabet, give this command:

```
PALPH)
```

The PALPH utility will respond by asking

WHAT DESTINATION FOR PALPH LISTING?

Respond with an acceptable CLI listfile name, such as **@LPT** for a line printer listing or @CONSOLE to display it on the screen.

PALPH will then list or display the current alphabetic characters, decimal point character, and currency symbol.

PFMT

Prints or Displays Formats

To print a format, give this command:

PFMT)

PFMT will ask:

NAME LISTINGS DESTINATION?

Give an acceptable CLI listfile name, such as @LPT or @CONSOLE.

PFMT will then ask for the names of the formats that you want to list. You may use angle brackets for your format names, such as BANKER<1,2,3>, but don't use templates or expansion forms that use parentheses.

End of Chapter

Chapter 9
Printing

The Idea system gives you several options for printing reports. The main method is to create printing records with a screen format/program module. Printing statements in the program send print records to the system COMMON file. You then print the records with the PRINTF utility, which uses a print format that you create with IFMT or WIFMT. The print format can be a copy of the screen format used to load the COMMON file, or you can tailor the print format to your specifications.

Another method for printing reports is to set up a DASHER printing terminal as a satellite (or slave) to the display terminal. The operator calls up a screen format/program module and completes the screen EDIT fields. Then, by pressing the PRINT key on the 6053 cursor pad, the operator sends a snapshot of the screen to the DASHER printer.

A third alternative is to use a DASHER printer as an Idea terminal. This method has some limitations; the most obvious is that the DASHER printer cannot print literal information to prompt the entry operator (although you can use DISPLAY fields as prompts).

Using PRINTF with a Print Format

To print reports using the PRINTF utility, follow these steps:

1. Create the screen input format and the printing output format.
2. Define the records used with both formats.
3. Write the IFPL program for the screen input format using the printing statements.
4. Compile the screen format and the program together, using the SYNTAX command.
5. Create the COMMON file using the DEFCOM utility.
6. Run the program, filling in the data fields.
7. Use the PRINTF utility with the printing format to print the report.

Creating Formats

To create the screen input format, use IFMT. Give the NONE response (just NEW LINE) to the prompt TYPE(H OR P OR NONE).

To create the printing format, use IFMT or WIFMT. Give the P response to the prompt TYPE(H OR P OR NONE). This allows you to use formats up to 80 lines long (60 with WIFMT), as well as to use the NEXT PAGE and PREV PAGE keys to move around while creating the format. The system will ask you for the length of the printed form. Regular line-printer paper is 66 lines long.

The P response tells the system that you will use the format for printing on a line printer; it therefore disables the attribute queries.

Designing the Records for Printing

When you design printing records, make sure that the record definition statements in the program match the field specifications in the printing format. This is crucial; the COMMON file has no way of delimiting fields. Thus, when PRINTF comes to the first field in the printing format, it takes as many bytes as the format specifies from the record in COMMON. For example, if the first field has a picture of six characters, PRINTF takes the first six bytes of the record from COMMON and inserts them on the printing line. It continues this process, field by field and line by line, until it empties the printing record.

If any field on the format doesn't match the associated field in the record, the fields will get out of sync. For example, if the first field in the record was only five bytes long and the format asked for six bytes, PRINTF would take the five bytes of the first field and the first byte of the second. Of course, this would throw off all following fields.

Writing the Program

The program that sends records to the COMMON file must contain the following statements:

```
RECORD FOR PRINTING IS recordname

INITIATE PRINTING IN printformatname

PRINT recordname USING printformatname

TERMINATE PRINTING USING printformatname
```

The RECORD FOR PRINTING statement begins a record definition block, just like the regular RECORD statement. You must include a LENGTH statement and the INCLUDES statements after the RECORD FOR PRINTING statement, and follow them with a STOP statement.

The INITIATE PRINTING statement begins the creation of the record in COMMON. After you give this statement, you can begin executing PRINT statements in the program.

The PRINT statement sends the specified record to the specified print format in COMMON, and the TERMINATE PRINTING statement ends the printing associated with the print format.

You cannot link a screen format to itself or to another format if you are using it to create print records. Linking can delete the print image if the system hasn't completed it. To repeat a format and return to a previous field, use a RETURN label statement; this will not log the terminal off like linking will.

Figure 9-1 shows a printing program and its associated formats.

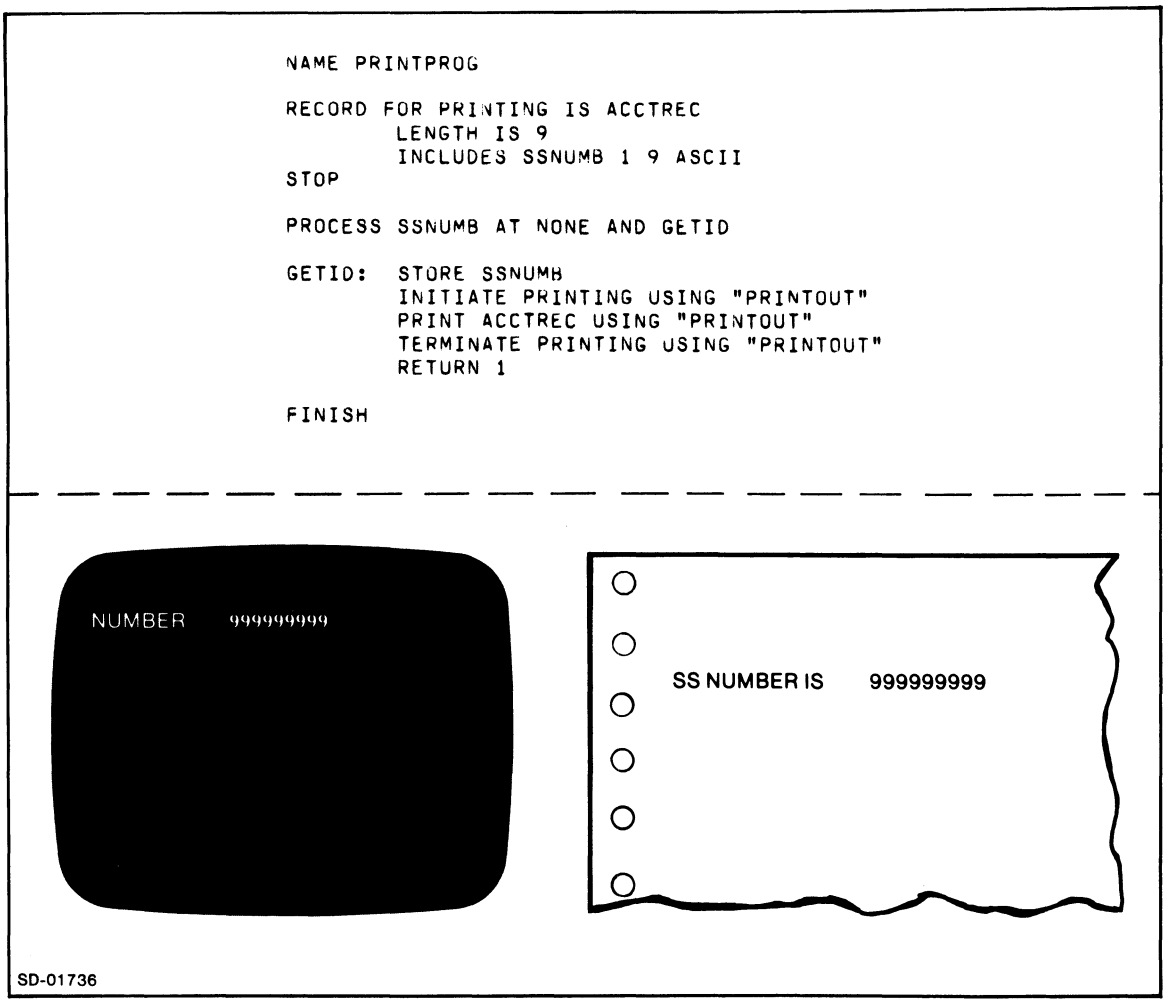


Figure 9-1. The Printing Program PRINTPROG.UP, the Screen Format PRINTPROG, and the Printing Format PRINTOUT

Creating the COMMON File

To create the system COMMON file, give this command:

DEFCOM)

This creates a standard COMMON file. We discuss the structure of COMMON and show you some ways that you can alter this basic structure in Appendix B.

When you first create COMMON, it is blank. After you use it with print records and formats, it contains the print record information. To delete old print records from COMMON, run DEFCOM again. The system will tell you that the COMMON file exists, and will ask if you wish to delete it. After you type a D to delete, DEFCOM creates a new blank COMMON file.

You can also use the PRINTF/D feature, which deletes records as it prints them.

Running the Input Program

You run the input format/program module from a local monitor, just as with any program.

Using PRINTF

To print the print records from COMMON, use the PRINTF utility. Give this command from the CLI:

PRINTF/A *[/D] [/L=listfile]* printformatname)

where:

printformatname	is the name of the format you used with the INITIATE PRINTING, PRINT, and TERMINATE PRINTING statements. You must give the print format the P option with IFMT or WIFMT.
/A	tells the system to print all records in COMMON regardless of which terminal supplied them.
/D	deletes records from COMMON as it prints them.
/L=listfile	lets you name a listfile other than @LPT.

Examples

PRINTF/A PATREP1)

Prints all records in COMMON using the format PATREP1.

PRINTF/A/D REPORTS)

Deletes all records as it prints them using the format REPORTS.

PRINTF/A/D/L=MYLIST SALESREP)

Sends the output to the file named MYLIST. Formats the data using the print format SALESREP. Deletes all records as it writes them to MYLIST.

To print only those records created by a specific terminal, give this version of the PRINTF command:

PRINTF *[/D] [/L=listfile]* printformatname console-number...)

To find the console-number, use the ISTATUS supervisory statement (see Chapter 10).

```
PRINTF SALESREP 01)
```

Prints all COMMON records created on terminal 01.

```
PRINTF/D CUSTACCT 01, 02, 05)
```

Prints and deletes all COMMON records created on consoles 01, 02, and 05.

Printing Scroll Fields

Printing scroll areas is a special case. The printing format must contain a pair of commercial at signs (@) around the scroll area. When PRINTF sees the first @ sign, it knows that it is printing a scroll area, and it will print scroll lines as long as the program continues to provide them.

However, PRINTF is also looking for the second @ sign to end the scroll area. But it never gets to the one on the format. Instead, you must supply one to end the scroll area. To do this you must create a record that contains an @ sign, and then print that record with a PRINT statement. The @ sign value terminates the scroll area.

For example, the program should contain a record such as

```
RECORD FOR PRINTING IS ENDSCROLL
     LENGTH IS 1
     INCLUDES "@" 1 1 ASCII
STOP
```

To terminate the scroll printing, you print the record ENDSCROLL just before giving the TERMINATE PRINTING statement for the scrolled format.

Inserting Your Own Form Feeds

PRINTF usually places a form feed after it has printed 62 lines. This is based on a 66-line form, skipping two lines at the top and two lines at the bottom. There are two ways to change this. You can specify a different form length when you create the printing format, or you can place your own form feeds in the format. To place your own form feeds within a format, place //FF// in the desired location while you are in LITERAL mode in IFMT.

Printing Headings After Form Feeds

To print headings after each form feed with PRINTF, enclose the heading in paired slashes (//). For example,

```
//This is a page heading//

@//This is a scroll heading. The @ sign begins the scroll area//
```

To use both page and scroll headings, you must start the printing format in page mode. Also, keep in mind that subsequent headings nullify previous ones.

PRINTF will print a page heading after each form feed. It will print a scroll heading if the scroll area runs over the page length. Thus, if your printing format had a form length of 66 lines and you scrolled 100 lines, PRINTF would print the headings on the second page.

Printing Screen Snapshots on a DASHER Printer

This form of printing is almost entirely hardware driven. To use it, you need a 6053 terminal with a printing board and a DASHER printing terminal attached as a satellite (or slave) printer.

To use this form of printing, the operator calls the screen input program, fills in the EDIT fields, and strikes the PRINT key on the cursor pad. Within the program, you must include a few statements that will print a snapshot of the screen when the operator strikes the PRINT key. These statements are:

```
        ON SCREEN tag

        REGISTER FIELD 99

              .

              .

              .

tag:  MESSAGE <10><21>

        RETURN USING FIELD
```

The statement **ON SCREEN tag** sends program execution to tag when the operator strikes the PRINT key. The **REGISTER FIELD 99** statement sets up the reserved word FIELD so that you use it with the RETURN USING statement. The statement **MESSAGE <10><21>** prints all data on the screen. The octal code 10 places the cursor at the home position, and the code 21 prints the screen.

To print only the variable data, use this MESSAGE statement:

```
MESSAGE <10><01>
```

Using a DASHER Printer as a Terminal

This method has some limitations, but you can work around these problems.

The DASHER printer cannot move its printing head backwards. Consequently, you can't display literals, and you can't use statements such as RETURN label. Also, you must be very careful when you send messages to a DASHER printer; the MESSAGE statement sends the printing head to line 24, and it can't get back from there.

So, to print label data for operator prompts, just use literal variables with DISPLAY fields and statements.

To repeat a program, link the format to itself instead of using a RETURN label, RETURN USING, or RESTART statement.

Another problem is that the DASHER doesn't know how long forms are. It uses a free-form length. You can set an artificial form length in page mode, by placing a dummy field on the next to the last line. For example, you would place a field at line 65 for a 66-line form.

Unfortunately, there is no similar method to use for scroll fields; you have to count the lines.

The fact that dummy fields will print on the form is another disadvantage. To make sure that the DASHER doesn't print anything at a dummy field, use zero suppress characters (Zs) for numeric dummy fields, or use Xs, which use the blank as the default character.

Some Sample Applications

We list the programs and formats discussed below in Appendix D.

Printing More Than One Report Per Page

PROBLEM:

How to output two or more print images to each printed page.

SOLUTION:

1. Link the printing format to itself.
2. Suppress the form feed on linking.
3. Make the format "form length" a multiple of the format length.

LISTINGS:

Screen format/program module DASHJR, DASHJR.UP. Output formats PAGEFMT and SCRLLFMT.

To print two PAGEFMTs per page we make PAGEFMT 31 lines long, link it to itself, and suppress the form feed on linking.

When creating PAGEFMT, we set the form length to 66 lines. If we set the length at anything less on a 66-line printer -- at 33 lines, for example -- PRINTF would issue a form feed to the printer as soon as it reached 33 lines. The printer would respond by going to the top of the next form, and this would leave the lower half of the form blank.

We derive the length of the format -- 31 lines -- with this formula:

format length = (form length-4)/# reports per page

If you know the format length, use this formula to find the form length:

form length = (# reports per page X format length) + 4

Find the number of reports per page by dividing the usable form length by the format length and discarding any fraction in the quotient. If the format length is 10 lines, then you can print 6 of them on one 66-line page, which contains 66 minus 4, or 62, usable lines. The form length specification should be (6 times 10) plus 4, or 64 lines.

When printing, PRINTF will issue a form feed when it uses up 64 lines. This keeps the printed reports in synchronization with the 66 lines of the form.

The figure 4 in the formulas reserves space for the two lines on the top and the two lines on the bottom of the form.

You can't do two-up printing by repeating the desired format on the lower half of the format; that is, by asking PRINTF to write one image on the top half of the format and a second on the bottom half. This violates its rule of having the data and the format end synchronously.

Figure 9-2 shows the output from our format PAGEFMT.

```
**************************************************************************
**************************************************************************

                          DATA GENERAL CORPORATION
                            4400 COMPUTER DRIVE
                           WESTBORO, MASSACHUSETTS


                            INVOICE NUMBER 000000
                               DATE: 05/12/77


PURCHASE ORDER NUMBER  3

CUSTOMER'S NAME:

PAUL PROTEUS
THE WORKS
ILIUM, NY

ITEM:  IDEA SYSTEM
UNIT PRICE:  $4,000
QUANTITY:  TWO (2)
AMOUNT:  $8,000

TOTAL AMOUNT THIS INVOICE:    $8,000


TERMS:  3 % TEN DAYS NET 30

**************************************************************************

                          DATA GENERAL CORPORATION
                            4400 COMPUTER DRIVE
                           WESTBORO, MASSACHUSETTS


                            INVOICE NUMBER 000000
                               DATE:  05/12/77


PURCHASE ORDER NUMBER  4

CUSTOMER'S NAME:

ELIOT ROSEWATER
GENERAL DELIVERY
ROSEWATER, IN

ITEM:  IDEA SYSTEM
UNIT PRICE:  $4,000
QUANTITY:  TWO (2)
AMOUNT:  $8,000

TOTAL AMOUNT THIS INVOICE: $8,000


TERMS:  3 % TEN DAYS NET 30

**************************************************************************
```

Figure 9-2. Printed Output Produced by PRINTF Using PAGEFMT

Generating Two Reports From a Single Idea Format

PROBLEM:

How to produce two different printed reports from the same Idea program.

SOLUTION:

1. Produce single-page reports of each transaction.
2. Produce a scrolled summary report of the terminal session.
3. Use a single print image for the entire terminal session covering all print records for both reports.

LISTING:

DASHDRVR, DASHDRVR.UP, PAGEFMT, SCRLLFMT

If you have more data than will fit on the printing format, you will trigger an error condition. PRINTF's default action for this error is to issue a form feed and restart the format. You can use this default to build different print reports in the same program.

The programs described below contain two reports. The first is a simple transaction report identical to that produced by DASHJR. The second is a summary report of the terminal session. It uses excerpts from each transaction to produce a scrolled summary of all transactions that were processed at the terminal session. Figures 9-3 and 9-4 contain these reports.

```
**************************************************************************
**************************************************************************


                          DATA GENERAL CORPORATION
                            4400 COMPUTER DRIVE
                           WESTBORO, MASSACHUSETTS


                            INVOICE NUMBER 000138
                               DATE: 05/12/77


PURCHASE ORDER NUMBER  000138

CUSTOMER'S NAME:

JEREMIAH JONES
33 SOUTH STREET
MISSOULA, MONTANA

ITEM:  IDEA SYSTEM
UNIT PRICE: $125,000
QUANTITY:  TWO (2)
AMOUNT: $250,000

TOTAL AMOUNT THIS INVOICE:    $250,000


TERMS:  3 % TEN DAYS NET 30

**************************************************************************
```

Figure 9-3. Printed Report of DASHDRVR Transaction Produced by Print Format PAGEFMT

```
                    DATA GENERAL CORPORATION

                    DGC DAILY INVOICE RECORD

                            05/12/77

COPIES: ACCOUNTING, PURCHASING, LEGAL, MANUFACTURING, MARKETING, FILE



 INVOICE NO. CUSTOMER'S NAME      ADDRESS           CITY, STATE, ZIP

000138     JEREMIAH JONES      33 SOUTH STREET      MISSOULA, MONTANA
000139     WENTWORTH PETERSON  111 MAIN STREET      TWIN FORKS, MAINE
000140     TALLULAH BANKHEAD   1 VINE STREET, APT 3 HOLLYWOOD, DAHLING
000141     VINCENT MALONE      27 CHEROKEE LANE     DULUTH, MINNESOTA

Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/ΔΔ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ
                    DGC DAILY INVOICE RECORD
 INVOICE NO. CUSTOMER'S NAME      ADDRESS           CITY, STATE, ZIP
000175     GEORGE JAMES        222 TURNPIKE ROAD    SILVER FALLS, MD.
000176     FRANCES MALONE      37 WASHINGTON ST.    BOSTON, MASS.
000177     GANDALF THE WHITE   RIVER RUNNING        MIDDLE EARTH
000178     FREDERICK JONES     ROUTE 33             ORANGE COUNTY, CA.

/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/Δ/

*************************************************************************
*************************************************************************
```

Figure 9-4. Summary Report Printed Out Using the Printing Format, SCRLLFMT

In order to produce a scrolled summary report, you must leave the print image open for the entire session. To keep it open, postpone the execution of the TERMINATE PRINTING statements for both reports until you log off. You must also remain within the program for the entire session; that is, you must repeat the screen with a RETURN 1, not by linking or RESTARTing. The latter two actions will restart the print image.

You cannot terminate the DASHJR-type transaction prematurely. To do so will reset the counter used by the print facility to supply the bottom keys of the print records in COMMON; i.e., both printing formats depend on the same counter. If you execute the associated print statements alternately, the print records for a particular format will have bottom keys that are either all odd or all even. This arrangement does not confuse PRINTF, which requires only that they be in ascending order. Both formats will be in ascending order if the print image is left open.

The printing format for the summary report, SCRLLFMT, uses the repeated heading facility of IFMT so that a page heading and a scroll heading repeat on each page of the report.

In order to use both page and scroll formats, you must include a field of page variable data at the beginning of the report. This initiates the format in page mode.

PAGEFMT prints the page records; this format is identical to the printing format for DASHJR. Here, however, the records print out one to a form. (For DASHJR, they print out two to a form.) This is a consequence of using a single huge print image for each terminal session. Thus, when the printing format PAGEFMT is satisfied, there is data left in the print image. PRINTF, therefore, takes its standard default for such a condition. That is, it issues a form feed and restarts the format.

You then use a single PRINTF command to print all records created during a terminal session, as well as all records created for the SCRLLFMT format. SCRLLFMT is linked to PAGEFMT via IFMT. So, when you tell PRINTF to print the SCRLLFMT records, it does so. Then it links to the page format and prints all the page records.

Be sure that you don't link PAGEFMT to itself. Such linking will disable the format for use by more than one terminal. Thus when SCRLLFMT prints out the summary report for a particular terminal and links to PAGEFMT, the latter prints out all the transaction reports for that terminal. If PAGEFMT is unlinked, PRINTF then looks for the next terminal. But if it is linked and the command PRINTF/A was issued, PRINTF will print out all PAGEFMT reports from all terminals before printing out the next SCRLLFMT summary report.

End of Chapter

Chapter 10
How to Load and Generate Idea

This chapter shows how to load the Idea system tape and how to generate the system. It also describes how to run a program from the local monitor.

If you are a system manager, you must complete steps 1, 2, 3, and 4 below. If you are a programmer, you may want to generate your own monitors for users (step 3). You will also bring up the local monitor to run your programs (step 5).

To get Idea up and running, you must perform the following tasks:

1. Load the tape containing the Idea system into the proper AOS directories.
2. Set the user search lists and the ACLs of these directories so that your users have access to them.
3. Generate your global and local Idea monitors with the IDEASG command.
4. Bring up the global monitor with the IDEA_UP command.
5. Bring up the local monitor, and run your program.

Before You Load the Tape

Before you load the tape containing the Idea system, you must:

1. Position yourself to the root directory (:).
2. Make sure that you are running under PID 2.
3. Set SUPERUSER ON.

Loading the Tape

The tape that contains the Idea system also contains a macro to help you bring up the Idea system. To load this macro, give the following command from the CLI (throughout this chapter we assume that you are using tape drive 0):

```
LOAD/V @MTA0:0 LOADIDEA.CLI)
```

To load Idea, you then execute the LOADIDEA.CLI macro.

Executing LOADIDEA

The LOADIDEA macro gives you two options when you create your Idea system. The default option creates a directory named :IDEASYSGEN and loads the contents of tape file IDEA_SYSGEN.DF into it. It also loads the contents of tape file IDEA_UTIL.DF into the :UTIL directory, and gives you the alternative of loading tape file IDEA_DIALOG.DF into :UTIL. IDEA_DIALOG.DF contains Idea's error messages and other dialogs.

The second option allows you to create and name two directories: one to contain IDEA_SYSGEN.DF and the other to contain IDEA_UTIL.DF and IDEA_DIALOG.DF.

To take the default option, give this command from the CLI:

LOADIDEA/DEF @MTA0:0)

To take the second option, creating your own directories, give this command:

LOADIDEA @MTA0:0 directoryname$_1$ directoryname$_2$)

Where:

directoryname$_1$ receives IDEASYSGEN.DF.

directoryname$_2$ receives IDEA_UTIL.DF and IDEA_DIALOG.DF.

After You Load the Tape

No matter which of the two LOADIDEA options you chose, you must set the ACLs of the two directories to allow all users READ and EXECUTE access (+,RE).

You must also make sure that all users have access to the local and global monitors. You can move the monitors to directory :UTIL, or you can include the two directories in the users' search lists. If you use the default directories, for instance, each user seach list must contain :UTIL and :IDEASYSGEN.

Generating the Idea Monitors

After you've loaded the tape, you generate the local and global monitors. The *global* monitor performs various supervisory functions for the system. It is a swappable process that usually uses few system resources. It is not attached to any console, but it does communicate with one designted console via system calls. This allows the supervisory console to perform other tasks as well.

The *local* monitor executes your programs. It exists as a process for each terminal running Idea. The local monitor consists of 24 shared pages and 2 unshared pages of memory. These figures do not take into account any format/program modules.

To create your global and local monitors, give this command from the CLI:

IDEASG *[/DIALOG] [/IPC] [monitorname/S] [loadmapname.LM/L]*)

This command begins a dialog with the system. To create the global and local monitors you must answer the various questions with numbers and uppercase letters. You may escape from this dialog by striking the ESC key.

If you give the default command IDEASG (NEW LINE), the system creates a global monitor named IDEA.PR and a local monitor named LIDEA.PR, as well as a global loadmap IDEA.LM and a local loadmap named LIDEA.LM.

You have the option of specifying your own *monitorname,* your own *loadmapname,* or both. These names must each contain 25 or fewer characters. You must specify the .LM extension with *loadmapname.*

The /DIALOG switch lets you save the IDEASG dialog in a file named monitorname.DL. You can then display the setting of the global monitor by using the ISYS command described in this chapter under "Bringing Up Global Idea."

The /IPC switch specifies that you want to use RCX70 as the IPC with SEND and RECEIVE statements. You must answer R to the IDEASG dialog question

DEFAULT PORT NAME (R = RCX70, NONE = NEW LINE)

Examples

```
IDEASG)
```

This command creates a global monitor named IDEA.PR and a local monitor named LIDEA.PR. It also creates a global load map IDEA.LM and a local load map LIDEA.LM.

```
IDEASG TOMSMON/S)
```

This command creates a global monitor TOMSMON.PR, a local monitor LTOMSMON.PR, a global load map IDEA.LM, and a local load map LIDEA.LM.

```
IDEASG CRAIGSMAP.LM/L)
```

This command creates a global monitor IDEA.PR, a local monitor LIDEA.PR, a global loadmap CRAIGSMAP.LM, and a local loadmap LCRAIGSMAP.LM.

```
IDEASG SAMSMON/S SAMSMON.LM/L)
```

This command creates a global monitor SAMSMON.PR, a local monitor LSAMSMON.PR, a global loadmap SAMSMAP.LM, and a local loadmap LSAMSMAP.LM.

The Sysgen Dialog

The IDEASG command begins a dialog in which you must answer the following questions. Use uppercase characters only.

TRANSACTION LOGGING (DISK = D, TAPE = T, NONE = N):

Answer D to log to disk, T to log to tape, or N for no logging.

NUMBER OF ACTIVE TERMINALS (DEFAULT IS 32):

Enter the maximum number of terminals that you want to run concurrently. The system maximum is 84.

FORMAT LIBRARY NAME (NONE = NEW LINE):

Enter the name of the format library if you wish to use one. You don't have to add the .FPL extension; IDEASG will do that automatically. If you specify a library, then you may use the formats in the library with this monitor only.

MAXIMUM PROGRAM SIZE (1 TO 8 BLOCKS):

A block is 1K words; enter a number large enough to contain your largest program. If you are using a format library, the largest program allowed is 7K.

TIME-OUT CONSTANT IN SECONDS (DEFAULT IS ZERO)

This feature stops infinite loops. The time-out constant is the amount of time that the monitor will allow an IFPL program for continuous execution. Timing begins when the monitor gives control to the program, and ends when the program returns control to the monitor. The program passes control to the monitor after each field transaction. If the program spends more than the amount of time you specify on a field transaction, the monitor will stop the program. If you select 0, timing is not done.

INITIAL FORMAT NAME (CAN BE OMITTED):

Specify an initial format name if you want the system to activate the format when an operator logs on. This is useful if you want to display an initial menu format at log on. The format name must be 10 or fewer characters long.

RECORD PASSING TYPE (D = DISK, C = CORE):

Core passing is faster, but it will pass a maximum of 512 bytes. To pass longer records, specify disk passing, which allows records up to 2040 bytes.

WILL YOU BE USING THE COMMON FILE (Y = YES, N = NO):

IDEASG asks this only if you specified CORE passing. Answer Y if you plan to use the COMMON file for printing or for any other purpose.

DEFAULT PORT NAME (R = RCX70, NONE = NEW LINE):

IDEASG asks this only if you specify the /IPC switch in the command line.

If you answered T for tape logging to the TRANSACTION LOGGING question, the system will ask the following questions:

NUMBER OF VOLUMES (1 TO 9)

This is the number of tape reels. The maximum is nine; there is no default.

LABEL TYPE (ANSI = A, IBM = I)

If you specify ANSI labels, the system sets the level number to 3. If you specify IBM labels, the system sets the level number to 2.

OWNER ID (CAN BE OMITTED)

An answer to this question is optional. By answering, you can assign an identification to each reel of tape.

VOLUME NAME (WILL BE NAME OF ALL VOLUMES)

The system recognizes tape reels by their volume name, not their tape drive destination. This name can be from 1 to 6 characters long.

AOS OPERATOR MESSAGE

This lets you include instructions to the AOS operator, who will mount the tape when you start up the global monitor.

FILE NAME

Logging goes to a tape file named volume:filename. You specify filename, which can be 1 to 7 characters long.

MAXIMUM RECORD BYTE LENGTH (MAXIMUM 4096)

Idea will write fixed length records of the length you specify here.

BLOCK SIZE (MUST BE A MULTIPLE OF RECORD SIZE)

Specify a block size that is a precise multiple of the record length you specified.

NUMBER OF BUFFERS

You must specify at least one buffer. By specifying two buffers you will improve response time, but you then run the risk of losing some of the log records if the system fails.

Bringing Up Global Idea

The next step is to bring up the global Idea process. Give this command:

IDEA_UP { *[/RES]* / *[/PRE]* } *[global_monitorname @CONx[/L] [/APPEND]]*

The optional switch /RES lets you bring up the global monitor as a resident process. The optional switch /PRE lets you bring up the global monitor as a preemptible process.

x is the number of the console you have designated as the system supervisory console. The global monitor will send various system messages to the supervisory console.

The /L switch sets LIST mode on; the global monitor will then display all ELOG messages as they reach the supervisory console.

The /APPEND switch will append new ELOG errors to the existing log rather than deleting the old log when you bring up the global monitor.

Note that you may use both the /L and /APPEND switches.

If you don't specify *monitorname* and console number in the command line, the system will ask for them.

Changing Tape Logging to Disk Logging

You may log records to a disk file instead of a tape file. To do this, you must first specify T for tape logging in the IDEASG dialog. Then, give the command

IDEA_UP)

from the CLI. The system will then ask you for the global monitor name, and for the number of the supervisory console. To the question

WHAT IS GLOBAL MONITOR NAME?

give a name with the switch /D=pathname attached. The pathname will be the logging file. You give the supervisory console number just as you would with any form of IDEA_UP.

For example,

WHAT IS GLOBAL MONITOR NAME? GIDEA/D=:UDD:BILL:RECLOG)

WHAT CONSOLE SHOULD OUTPUT GO TO? 04)

If you do not specify a full pathname, the system will place the file in :PER.

Supervisory Console Commands

You can give these Idea commands from the supervisory console: IABORT, IBYE, IELOG, IENABLE, IHELP, IINHIB, ILIST, IMESSAGE, IKMSG, ISTATUS, ISYS. Table 10-1 lists these commands.

Table 10-1. The Supervisory Commands

Command	Action
IABORT nn	Shuts down the local Idea monitor specified by nn (to determine nn, give the ISTATUS command).
IBYE	Shuts down the Idea system if there are no local Idea monitors running. If local Idea processes are present, the system displays a message, and the global monitor does not terminate.
IELOG nn	Displays the most recent nn entries to ELOG. If nn is larger than the number of lines in ELOG, the system displays the entire contents.
IENABLE nn	Enables the local Idea log-on process at the console numbered nn. To enable all consoles, specify + for nn.
IHELP	Displays a list of all global Idea commands; i.e., those listed here.
IINHIB nn	Inhibits the local Idea log-on process at the console numbered nn (the opposite of IENABLE). To inhibit the log-on process at all terminals, specify + for nn.
ILIST arg	Specify ON, OFF, or ? for arg. ON sets list mode on, OFF sets it to off, and ? displays the current setting. If list mode is set ON, then the system displays all ELOG messages as it receives them.
IMESSAGE	Displays the next line that you type on line 24 of all local Idea consoles.
IKMSG	Cancels a message sent with IMESSAGE.
ISTATUS	Returns a list of logged-on local Ideas, as well as a list of currently inhibited console numbers.
ISTATUS nn	Returns complete log-on statistics for the local Idea process at console nn (if it exists), as well as a list of inhibited console numbers.
ISTATUS +	Returns complete log-on statistics for all active local Idea processes, as well as a list of inhibited console numbers.
ISYS	Displays the characteristics of the current global monitor if you specified the /DIALOG switch in the IDEASG dialog.

If you specified that you will be using the COMMON file, you must create it with DEFCOM before you run the local monitor. To run the local monitor, give this command from the CLI:

X localmonitorname $\left\{\begin{matrix}[/RES] \\ [/PRE]\end{matrix}\right\}$)

The optional switches allow you to bring up the monitor as resident (/RES) or preemptible (/PRE). (Be sure you are privileged for this option -- the system will not generate an error message if you are not.)

For example, if you generated Idea with the default names, you would give this command:

X LIDEA)

The local monitor will ask for an ID (optional). Then, if the person who generated the system specified an initial format, the local monitor will display that format. If not, the local monitor asks for the name of the desired format.

After you give the format name, the monitor asks if you would like to see the data type of the current screen field. Type Y NEW LINE for yes; type NEW LINE for no.

The monitor then displays the format on the screen, ready to accept input into the EDIT fields.

System Considerations of the Local Monitor

If you are operating in an environment with a small number of terminals, you will probably want the operators to run their monitors as described above, from the CLI.

If you are in a production environment, however, this method can cause system overhead problems, since each monitor will be an AOS process. In such an environment, you may want to set up the local monitor as the initial AOS process, called up when the operator logs on the AOS system. To do this, use the AOS Profile Editor (PREDITOR). You may create one Idea user profile for all operators, or you may create a separate profile for each. The latter method uses the AOS file protection facilities.

When you edit the user's profile, change the initial program from its current setting (the default is :CLI.PR) to the full pathname of the local monitor. For example:

PROGRAM [:CLI.PR] CHANGE (Y OR NL) Y)

NEW (2-63 CHARS): :UTIL:LIDEA.PR)

If you want the local monitor as the initial process and also want it to be resident or preemptible, you must change the user profile's INITIAL IPC. You must give a complete pathname to file SLASHRES for resident or to file SLASHPRE for preemptible. SLASHRES and SLASHPRE both assume that you are using the default local monitor name LIDEA, so you must edit the files if you give your monitor another name.

Table 10-2 lists the functions performed by the function keys, which are labeled by the templates (the side marked Idea). Operators can use these functions when entering data.

Table 10-2. The Operator Data Entry Special Function Keys

Function Key	Meaning
LOG OFF	Logs operator off.
END DATA	Ends screen input to current screen. Links to format named in IFMT, if any; otherwise, asks for a new format.
REPEAT PAGE	Deletes operator entries to EDIT fields; then, redisplays current format.
CHANGE MODE	Terminates scroll mode.
ERASE FIELD	Erases entry in current EDIT field.
DUP FIELD	Duplicates field in scroll line from corresponding field on previous line.
BACK TAB (Unmarked key on 6053 cursor pad)	Moves cursor back to first character of current field. Then, moves successively back to first character of preceding fields.
NEGATE SIGN	Makes a signed number negative.
MINUS ENTER	Makes a signed number negative and enters it.
ENTER	Enters data (works just like NEW LINE).

End of Chapter

Appendix A
Converting Programs Between AOS and RDOS

Converting from RDOS to AOS

To convert programs developed under RDOS to AOS, follow these steps:

1. Under RDOS, dump the formats to tape using this command:

   ```
   DUMP/V MT0:0 formatname.<,VS,FS>
   ```

2. Under RDOS, dump the programs to tape using this command:

   ```
   DUMP/V MT0:1 programname.UP
   ```

   (Note that you should use different tape files for the formats and the programs.)

3. Use the AOS utility RDOS to load the tapes. For the formats, use

   ```
   XEQ RDOS LOAD/V @MTA0:0
   ```

   Do not use the /C switch for the formats. However, you must use the /C switch with the program files, as in the following:

   ```
   XEQ RDOS LOAD/V @MTA0:0 +/C
   ```

   The /C switch converts carriage returns to NEW LINEs.

4. Compile your formats and programs.

Converting from AOS to RDOS

There are two methods to do this.

Method 1

1. Dump the format files and program files to tape, using the AOS utility RDOS. Attach the /C switch to all program files.

2. Compile the formats and programs.

Formats created with IFMT revision 2.00 or later will not work. Also, AOS programs with more than 40 fields will not work.

Method 2

1. Use the RDOSYNTAX command to compile the formats and programs.

   The syntax of the RDOSYNTAX command is

   RDOSYNTAX *[/L] [/A] [/W] [/N]* **formatname programname**

   Where:

   formatname is the name of a valid format in the current directory.

   programname is the name of an IFPL program that exists on your disk. If you use formatname.UP as your programname, you don't have to include programname in the command line.

   /L Gives you a line-printer listing of the source text.

   /A Gives you a line-printer listing of the source text plus a line-printer listing of the assembly language statements that the compiler generates.

   /W Suppresses nonfatal error messages; we recommend that you use this only after initial syntaxing.

   /N Compiles the program, but doesn't assemble or load it. It also displays error messages on the terminal screen.

2. Dump the files to tape using the AOS utility RDOS. Don't use /C with the screen format files; do use /C with the program files.

3. Call the monitor and run the program.

End of Appendix

Appendix B
The COMMON File

The Idea system COMMON file is a three-level INFOS file. When you create it using the Idea utility, DEFCOM, it has the parameters of the ICREATE dialog shown in Figure B-1.

```
ICREATE/T=COMMONER

 ****** INFOS FILE CREATION      5/22/79 13:34:7 ******

NAME OF FILE TO BE CREATED: COMMON
ACCESS METHOD (I=ISAM, D=DBAM) [D]:

 ****** DEFINE INDEX FILE ******

        MAXIMUM NUMBER OF INDEX LEVELS [2]: 3
        PAGE SIZE (BYTES) [2048]:
        PARTIAL RECORD LENGTH [0]:
        ROOT NODE SIZE [2042]:
        MAXIMUM KEY LENGTH [255]: 13
        ALLOW DUPLICATE KEYS IN THIS INDEX? (Y OR [N]):
        ENABLE SPACE MANAGEMENT? (Y OR [N]):

         ****** DEFINE INDEX VOLUME(S) ******

        NUMBER OF VOLUMES TO DEFINE [1]:
        VOLUME 1 NAME [VOL01]:
                SPECIFY MAXIMUM SIZE? (Y OR [N]):
                SPECIFY FILE ELEMENT SIZE? (Y OR [N]):

 ****** DEFINE DATABASE FILE ******

        DATABASE FILE NAME [COMMON.DB]:
        PAGE SIZE (BYTES) [2048]:
        ENABLE SPACE MANAGEMENT? (Y OR [N]):

         ****** DEFINE DATABASE VOLUME(S) ******

        NUMBER OF VOLUMES TO DEFINE [1]:
        VOLUME 1 NAME [VOL01]:
                SPECIFY MAXIMUM SIZE? (Y OR [N]):
                SPECIFY FILE ELEMENT SIZE? (Y OR [N]):
```

Figure B-1. The ICREATE Parameters Used by DEFCOM

You can customize COMMON for a particular installation by using ICREATE to create an index file named COMMON and a database file named COMMON.DB. To do this, you must delete the existing COMMON file with this command:

```
IDELETE COMMON.DB)
```

The COMMON Print Facility

Each execution of a PRINT statement in an IFPL program generates a record in the COMMON file. The record thus generated corresponds to the record description block referenced in the IFPL program. The record is indexed by three keys, as shown in Table B-1.

A print image consists of a set of *n* records whose level-two binary count runs from 1 to *n*, but whose level-zero key, level-one key, and duplicates count are identical. You can initiate such a print image with an INITIATE PRINTING statement and terminate it with a TERMINATE PRINTING statement. Multiple print images with identical print formats and CRTs are distinguished by the duplicates count at level one.

You may output print images that are in COMMON to a line printer by issuing a properly formatted PRINTF command to the CLI.

Table B-1. Keys Used for Print Records in the COMMON File

Level	Key	Length	Type	Explanation
0	KEYNAME	1-13	ASCII	KEYNAME is the name of the print format.
1	CRT #, duplicates allowed	2	ASCII	CRT # is the system # assigned to the CRT that executed the print statement.
2	binary count	2	binary	Binary count starts at 1 and increments 1 for each additional print statement that is executed under a particular duplicate count at level 1.

Figure B-2 shows what COMMON looks like to an IFPL program. At the top level (level 0), the key type is ASCII with a maximum length of 13 bytes or characters. The key value used at this level is the print format name; that is, the actual format name that you will use with a CLI command of the following type to obtain line printer output:

PRINTF/A format)

No record is associated with level 0.

At the second level (level 1), the key type is again ASCII with a maximum length of two bytes. In this case, duplicate occurrences are permitted. The key value used is the system number of the terminal from which the print statement is executed. A record is associated with this level; it is discussed below.

At the third level (level 2), the key type is binary with a length of two bytes. The third-level key values in a print image comprise a series from 1 to *n*.

```
*FILE DESCRIPTION

FILE COMMON
KEY FOR COMMON IS 13 ASCII
SUBINDEX FOR COMMON IS LEVEL1
KEY FOR LEVEL IS 2 ASCII
DUPLICATES ARE COUNTED IN DUPCOUNT
SUBINDEX FOR LEVEL1 IS LEVEL2
KEY FOR LEVEL2 IS 2 BINARY
RECORD FOR LEVEL1 IS LEVEL 1REC
      LENGTH IS 2
      INCLUDES PRINTFLAG 1 2 BINARY
      STOP

RECORD FOR LEVEL2 IS LEVEL2REC
COPY RECORD
      *RECORD IS THE RECORD FOR PRINTING DESCRIPTION
      *IN THE USER IFPL PROGRAM

STOP
```

Figure B-2. An IFPL View of COMMON

When an IFPL program at a particular CRT executes an INITIATE PRINTING statement, it initiates a print image. Such an execution supplies two keys: the format name from the INITIATE PRINTING USING key statement (the format name is the key), and the CRT number, which IMON maintains as a system value.

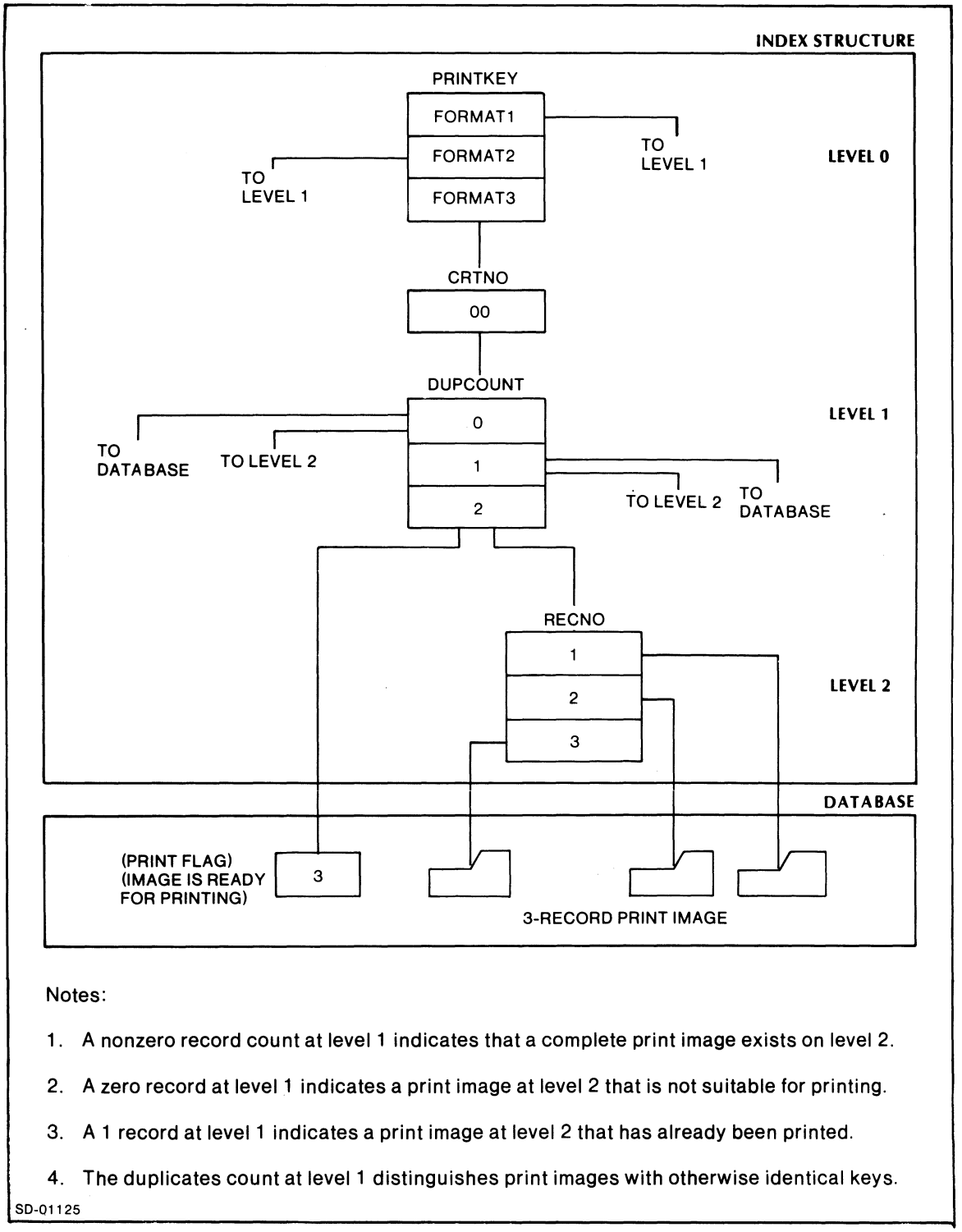
If such execution is the first to use those two keys since COMMON was defined with DEFCOM, the system assigns a duplicates count of 0 to the CRT number. The system will key the next PRINT statement that the program executes using the same print format by format, crt #, 1; the second by format, crt #, 2; and so on.

The execution of a TERMINATE PRINTING statement, resets the binary count at level 2. The subsequent execution of an INITIATE PRINTING statement will increment the duplicates count at level 1. Level-two records written out under the new duplicates count will again range from 1 to *n*.

This arrangement permits the existence in COMMON of multiple print images with keys that are identical except for the duplicates count. Level 1 contains a 2-byte binary record which is used as a print flag. It is keyed by the print format name and the CRT number that generated it, together with its duplicates count. An INITIATE PRINTING statement will set this record to 0. A TERMINATE PRINTING statement will rewrite the record so that it equals the number of records in the print image. A 0 in this record may thus be a flag that means "print image being built". A nonzero number means "print image is ready to output". PRINTF will again rewrite this record, setting it to - 1 to indicate the record has been printed.

You can delete COMMON print records which you no longer need by using either the /D switch on PRINTF (which deletes records as they are printed) or DEFCOM (which deletes everything in the file and rebuilds it).

The COMMON printing facility is presented graphically in Figure B-3.



Notes:

1. A nonzero record count at level 1 indicates that a complete print image exists on level 2.
2. A zero record at level 1 indicates a print image at level 2 that is not suitable for printing.
3. A 1 record at level 1 indicates a print image at level 2 that has already been printed.
4. The duplicates count at level 1 distinguishes print images with otherwise identical keys.

SD-01125

Figure B-3. COMMON Printing Facility

The COMMON Passing Facility

The passing facility uses the same INFOS file as the printing facility; its structure is shown in Figure B-4. Passing uses only two levels and does not use duplicates. Recapping the previous description, the top level of COMMON has an ASCII key with a maximum length of 13 characters. The level-one key is also ASCII and has a maximum length of two bytes.

Normally, you create COMMON with DEFCOM, the Idea utility. DEFCOM sets up 32 blank records for passing. At the top level, the system writes the key ??PASSING?? when you create the file; there is no record at this level. At the second level (level 1), the terminal number of each terminal defined in the system is written as an ASCII key at file creation time. Whenever the system executes a PASS statement, it performs a file rewrite at level 1, using the key ??PASSING?? and the terminal number of the program that is executing the PASS statement. The record that is rewritten is the one named in the PASS statement and described in the associated RECORD FOR PASSING description block. It may be a maximum of 1016 bytes long.

If you use the passing records for any other purpose than passing, then you must describe COMMON in your program. The IFPL description of COMMON, as used for passing, is as follows:

```
FILES COMMON
KEY FOR COMMON IS 13 ASCII
SUBINDEX FOR COMMON IS LEVEL 1
KEY FOR LEVEL 1 IS 2 ASCII
RECORD FOR LEVEL 1 IS PASSREC
LENGTH IS 15
INCLUDES POINTER 1 6 ASCII
STOP
```

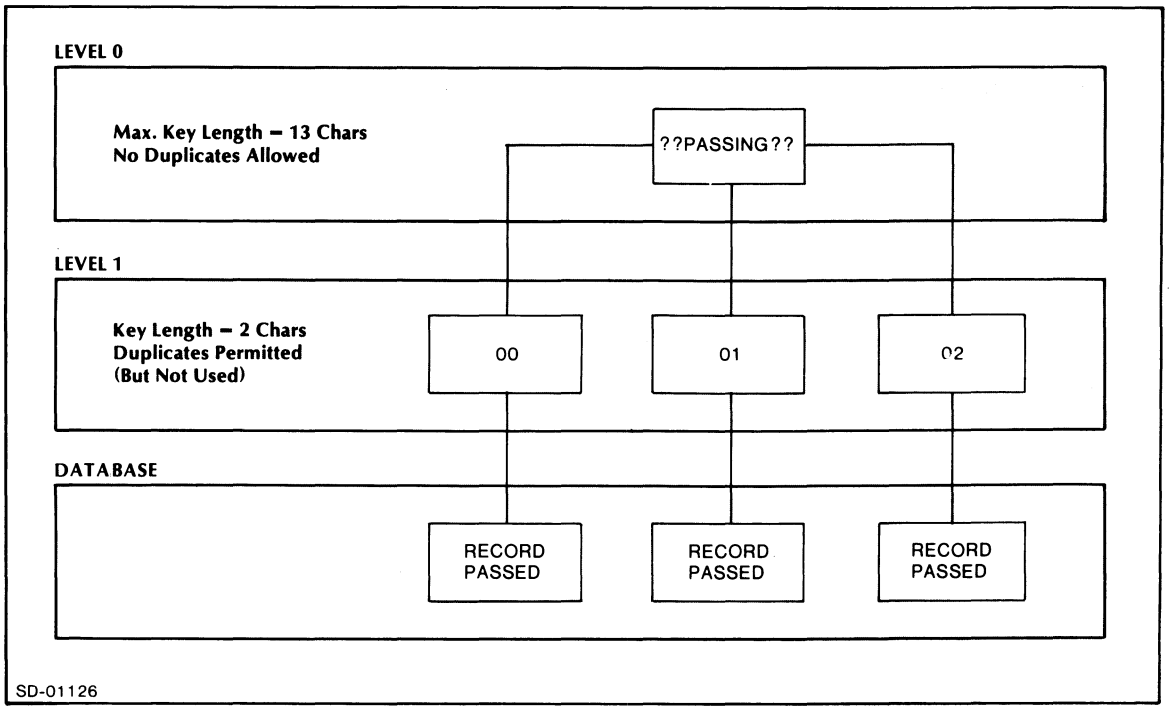


Figure B-4. The COMMON Passing Facility

Inspecting COMMON with Idea

You use an Idea screen to see the structure of the COMMON print file. Such a screen can read the file and present both contents and keys in a single coherent display. The screen SHOWME, illustrated in Figure B-5 and B-6, is such a display; it will read any COMMON print record.

SHOWME has effectively doubled the Idea scroll buffer capacity, 504 bytes, by leaving scroll mode and immediately re-entering it. It accomplishes this by using adjacent scroll area delimiters, @ signs, on the screen. Of course, only the second area actually scrolls; however, in this application, that is sufficient.

```
PRINT OF FORMAT: SHOWME

                                SHOWME READS PRINT RECORDS FROM THE COMMON FILE

X

ENTER TOP KEY  XXXXXXXXXXXXX   X
CRT# IS 99    OCCURRENCE COUNT IS ZZZ9   PRINT FLAG IS ZZZ9

INITIAL 3-LEVEL KEY IS XXXXXXXXXXXXX, 99 (DUP COUNT = ZZZ9), ZZZZ

  REC #   RECORD (FIRST 70 BYTES ONLY OF THE LEVEL 2 RECORD)
@ZZZZ  XXXXXXXXXXXXXXXXX//XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX X

@@ZZZZ  XXXXXXXXXXXXXXXXX//XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX X

    @ANOTHER PRINT IMAGE?  X    ANOTHER TOP KEY?  X

16:24:44 08/11/77
PRINT OF FORMAT: SHOWME
     Field #  Description  Disp  Edit  Output  Auto-  Req.  Full  Auto-  Sec
                                                Dupe   Entry  Field  Entry

      1  1     X(1)        *
      2  2     X(13)             *
      3  3     X(1)        *
      4  4     9(2)        *
      5  5     9(4)        *
      6  6     9(4)        *
      7  7     X(13)       *
      8  8     9(2)        *
      9  9     9(4)        *
     10  10    9(4)        *
     11  11    9(4)        *
     12  12    X(70)       *
     13  13    X(1)              *
     14  14    9(4)        *
     15  15    X(70)       *
     16  16    X(1)              *
     17  17    X(1)              *
     18  18    X(1)              *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 231
```

Figure B-5. Using SHOWME to Inspect the COMMON File

Suppose that a program exists which writes the name BIGFOOT to COMMON in oversize characters, and uses FTITLE as the printing format. If the writing terminal has the system value 01 and if this is the first time the program has been used on that terminal and if the print image has not been printed with PRINTF, then the SHOWME display will be as shown in Figure B-6.

```
        SHOWME READS PRINT RECORDS FROM THE COMMON FILE
        ***********************************************

ENTER TOP KEY PTITLE
CRT # IS 01   OCCURRENCE COUNT IS  0  PRINT FLAG IS 9


INITIAL 3-LEVEL KEY IS  PTITLE,01 (DUP COUNT = 0),   1


REC#        RECORD   (FIRST 70 BYTES ONLY OF THE LEVEL 2 RECORD)
 1  BBBBBB   IIIIIII    GGGGG   FFFFFFF     000       000      TTTTTTT
 2  BB  BB     II      GG   GG  FF         00 00     00 00       TT
 3  BB  BB     II      GG       FF        00   00   00   00      TT
 4  BBBBB      II      GG GGGG  FFFFF     00   00   00   00      TT
 5  BB  BB     II      GG   GG  FF        00   00   00   00      TT
 6  BB  BB     II      GG   GG  FF         00 00     00 00       TT
 7  BBBBB    IIIIIII    GGGGG   FF          000       000       TT
 8  @
 9  07/11/78

            ANOTHER PRINT IMAGE? - ANOTHER TOP KEY?
```

Figure B-6. Using BIGFOOT and PTITLE

End of Appendix

Appendix C
The Transaction File TRANS

The transaction logging file TRANS is a multilevel INFOS file. Any format can use TRANS with or without an associated program. TRANS accepts the contents of any screen field with the OUTPUT attribute.

To use TRANS, you must use a local monitor with the DISK LOGGING attribute specified during the IDEASG dialog.

The monitor writes fields to TRANS after it completes a page or scroll group. Each such writing constitutes a record.

The transaction buffer is 200 bytes long. This, then, is the maximum number of data characters with the OUTPUT attribute that a screen group can contain.

Creating TRANS

To create the TRANS file, give this command from the CLI:

```
ICREATE/B=TRANSACTION.FF)
```

We have supplied the INFOS trail file, TRANSACTION.FF, with the system tape. Figure C-1 shows its contents.

To get rid of old TRANS values, you must delete the TRANS file. You then give the ICREATE/B=TRANSACTION.FF command to build a new, blank one.

To delete TRANS, give this command from the CLI:

```
IDELETE TRANS.DB)
```

```
NAME OF FILE TO BE CREATED: TRANS
ACCESS METHOD (I=ISAM, D=DBAM) [D]: D

 ****** DEFINE INDEX FILE ******

        MAXIMUM NUMBER OF INDEX LEVELS [2]: 5
        PAGE SIZE (BYTES) [2048]:
        PARTIAL RECORD LENGTH [0]:
        ROOT NODE SIZE [2042]:
        MAXIMUM KEY LENGTH [255]: 14
        ALLOW DUPLICATE KEYS IN THIS INDEX? (Y OR [N]):
        ENABLE SPACE MANAGEMENT? (Y OR [N]):

         ****** DEFINE INDEX VOLUME(S) ******

        NUMBER OF VOLUMES TO DEFINE [1]:
        VOLUME 1 NAME [VOL01]:
                SPECIFY MAXIMUM SIZE? (Y OR [N]):
                SPECIFY FILE ELEMENT SIZE? (Y OR [N]):

 ****** DEFINE DATABASE FILE ******

        DATABASE FILE NAME [TRANS.DB]:
        PAGE SIZE (BYTES) [2048]:
        ENABLE SPACE MANAGEMENT? (Y OR [N]):

         ****** DEFINE DATABASE VOLUME(S) ******

        NUMBER OF VOLUMES TO DEFINE [1]:
        VOLUME 1 NAME [VOL01]:
                SPECIFY MAXIMUM SIZE? (Y OR [N]):
                SPECIFY FILE ELEMENT SIZE? (Y OR [N]):
```

Figure C-1. The Contents of TRANSACTION.FF

The Structure of TRANS

Table C-1 shows the internal structure of TRANS.

Table C-1. The Structure of the TRANS File

Key Formats				Records		
Level	**Key**	**Length**	**Type**	**Contents**	**Length**	**Type**
0	Crt[1]	2	B	Yr/Mo/Day[6]	6	A
1	Batch[2]	3	A	Hr/Min/ID[7]	14	A
2	Format[3]	10	A	(none)[8]		
3	10,20,10(n)[4]	2	B	Group Header[9]	10	B
4	10,20,10(n)[5]	2	B	Transaction[10]	1-200	A

A - ASCII B - binary

Notes:

1. The terminal number; corresponds to the reserved word, CRT.
2. The batch value entered by the operator when logging on the Operator's Console; corresponds to the reserved word, BATCH.

Table C-1. The Structure of the TRANS File (continued)

Notes (continued):

3. The name of the format used for data logging, left-justified and blank-filled as necessary to get a 10-byte key.

4. The Group Header key. The key sequence starts at 10 for a particular format, and is incremented by 10 for each group header encountered; i.e., for each change from page mode to scroll mode and vice versa. The system continues this sequence by incrementing the last key used by 20 each time a format is re-executed.

   Before the format is re-executed, the system writes a dummy record to TRANS to separate the two groups of records. Its key is 10 more than the last key used before the format was re-executed; i.e., it continues the key sequence unbroken.

5. The bottom key for the transaction record. The key sequence starts at 10 and is incremented by 10 for each transaction record written. The sequence starts at 10 each time the group changes.

6. The system year, month, and day. They correspond to the reserved words, YEAR, MONTH, and DAY.

7. The system time and the operator's identification. The system time corresponds to the reserved words, HOURS and MINS. The operator's ID corresponds to the reserved word, PASSWORD.

8. No record is written here. The only item of interest, the format name, is already contained as the value of the key.

9. This record is a modified format of the group header contained in the format.VS file. The meaning of its 10 binary bytes is as follows:

Byte #	Contents
1	Mode (Page = O, Scroll = 128_{10})
2	Entries. This is the number of fields in the group, irrespective of whether they have the OUTPUT attributes.
3,4	Sum of field lengths in the group, irrespective of whether they have the OUTPUT attributes.
5,6	Starting row for the group. Only meaningful for scroll groups.
7	Group sequence number (the number of the group on the screen); starts at 0. The system increments it by 1 each time it encounters a group header, and resets it to 0 each time the format is executed.
8	Total number of rows; only meaningful for a scroll group. It tells the monitor when to start scrolling.
9,10	TRANS record length. It is the sum of all output field lengths in the format.

   The dummy header record that stands at the end of any header sequence and separates sequences caused by format re-execution is laid out as above; the first two full words (four bytes) are set to -1 and the other words are set to 0.

10. This is the TRANS record. It contains all the fields designated as OUTPUT in one page group or one scroll line.

Displaying TRANS Contents

You can display the contents of the TRANS file with an Idea format and program. The format and program shown in Figure C-2, READTRAN, will read any TRANS file and display its contents.

```
12:32:08 03/10/78
PRINT OF FORMAT: READTRAN
                    TRANSACTION DISPLAY

CRT :     99                                        99-99-99

BATCH :   XXX       99:99                     ID:   XXXXXXXXXX

FORMAT :  XXXXXXXXXX  (DEFAULT IS TRANSFILE)

GROUP :   99          ENTRIES :                      9
HEADER                SUM OF FIELDS :              +++9
KEY                   TRAN RECORD LENGTH :          ZZ9
                      GROUP NUMBER :               ZZZ9

STRIKE ANY KEY TO SEE NEXT RECORD: X
@
PART 1 :  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
PART 2 :
PART 3 :
PART 4 :
PART 5 :
@
STRIKE ANY KEY TO SEE NEXT GROUP: X



**********************************************************************************************
**********************************************************************************************

PRINT OF FORMAT: READTRAN
     PHYS./LOG.                                          AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE    ENTRY  FIELD  ENTRY  SEC
        1   1  9(2)               *                                     *
        2   2  9(2)          *
        3   3  9(2)          *
        4   4  9(2)          *
        5   5  X(3)               *                                     *
        6   6  9(2)          *
        7   7  9(2)          *
        8   8  X(10)         *
        9   9  X(10)         *    *
        10  10 9(2)          *
        11  11 9(1)          *
        12  12 S9(3)         *
        13  13 9(3)          *
        14  14 9(4)          *
        15  15 X(1)               *                                     *
        16  16 X(45)         *
        17  17 X(1)               *                                     *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 22
```

Figure C-2. READTRAN

```
ACS SYNTAX REV 01.01     READTRAN.VS   READTRAN.UP  12:33:4   3/10/78


1               NAME    READTRAN.UP
2
3       *
4       * THE PURPOSE OF THIS PROGRAM IS TO DISPLAY THE
5       * CONTENTS OF AN IDEA TRANS FILE.
6       *
7       * THE MAKEUP OF THE TRANS FILE CAN BE
8       * SEEN BY INSPECTING THE INDEX STRUCTURE,
9       * KEYS, AND RECORD FORMATS SHOWN BELOW.
10      *
11      * THE PROGRAM IS INITIALIZED TO READ
12      * TRANS RECORDS WRITTEN BY THE FORMAT
13      * TRANSFILE. THIS CAN BE OVERRIDDEN BY
14      * THE OPERATOR TO READ ANY TRANS FILE
15      * RECORD.
16      *
17      FILE    TRANS
18
19      SUBINDEX FOR TRANS IS BATCH
20      SUBINDEX FOR BATCH IS FORMAT
21      SUBINDEX FOR FORMAT IS GROUP
22      SUBINDEX FOR GROUP IS LINE
23
24      KEY FOR TRANS IS 2 BINARY
25      KEY FOR BATCH IS 3 ASCII
26      KEY FOR FORMAT IS 10 ASCII
27      KEY FOR GROUP IS 2 BINARY
28      KEY FOR LINE IS 2 BINARY
29
30
31      RECORD FOR TRANS IS DAY.REC
32              LENGTH IS 6
33              INCLUDES        YRS             1 2 ASCII
34              INCLUDES        MTH             3 2 ASCII
35              INCLUDES        DYS             5 2 ASCII
36              STOP
37
38      RECORD FOR BATCH IS TIME.REC
39              LENGTH IS 14
40              INCLUDES        HRS             1 2  ASCII
41              INCLUDES        MINUTES         3 2  ASCII
42              INCLUDES        IDENT           5 10 ASCII
43              STOP
44
45      RECORD FOR GROUP IS GROUP.HEAD
46              LENGTH IS 10
47              INCLUDES        ENTRIES         2 1  BINARY
48              INCLUDES        SUM.FL          3 2  BINARY
49              INCLUDES        TRAN.LN         9 2  BINARY
50              INCLUDES        GROUP.NO        7 1  BINARY
51              STOP
52
53      RECORD FOR LINE IS TRAN.REC
54              LENGTH 200
55              INCLUDES        PART1             1  40  ASCII
56              INCLUDES        PART2            41  40  ASCII
57              INCLUDES        PART3            81  40  ASCII
58              INCLUDES        PART4           121  40  ASCII
59              INCLUDES        PART5           161  40  ASCII
60              STOP
61
```

Figure C-2. READTRAN (continued)

```
62      RECORD FOR FORMAT IS NO.REC
63              LENGTH IS 0
64              STOP
65
66      REGISTER PARTS          S999
67      REGISTER COUNTER        9
68      REGISTER GRP            99      10
69      REGISTER LNNO           99      10
70      REGISTER TEN            99      10
71      REGISTER NOREC          99      23
72      REGISTER NO.ENTRY       XXX
73      REGISTER SPACE          X(10)
74      REGISTER PART2          X(40)
75      REGISTER PART3          X(40)
76      REGISTER PART4          X(40)
77      REGISTER PART5          X(40)
78      REGISTER ZERO           9
79      REGISTER FLAG           9(2)
80      REGISTER FIELD 99
81
82
83
84      A1#     PROCESS CRT.NO AT NONE AND EC
85      A2#     PROCESS YRS AT DY AND NONE
86      A3#     PROCESS MTH AT DM AND NONE
87      A4#     PROCESS DYS AT DD AND NONE
88      A5#     PROCESS BATCH.NO AT NONE AND EB
89      A6#     PROCESS HRS AT DH AND NONE
90      A7#     PROCESS MINUTES AT DMI AND NONE
91      A8#     PROCESS IDENT AT DI AND NONE
92      A9#     PROCESS FMT.NAME AT DF AND EF
93      A10#    PROCESS FILLER AT DG AND NONE
94      A11#    PROCESS ENTRIES AT DE AND NONE
95      A12#    PROCESS SUM.FL AT DSFL AND NONE
96      A13#    PROCESS TRAN.LN AT DTLN AND NONE
97      A14#    PROCESS GROUP.NO AT DGN AND NONE
98      A15#    PROCESS FILLER AT NONE AND DL
99      A16#    PROCESS PART1 AT DP1 AND NONE
100     A17#    PROCESS FILLER AT NONE AND NEXT
101
102     ****
103     * A1
104     ****
105     EC:     STORE CRT.NO
106             MOVE "0" FLAG
107             FIND DAY.REC USING CRT.NO
108             ON-IOERR NO.CRT
109             RETURN
110
111     ****
112     * A2
113     ****
114     DY:     DISPLAY YRS
115             RETURN
116
117     ****
118     * A3
119     ****
120     DM:     DISPLAY MTH
121             RETURN
122
123     ****
124     * A4
125     ****
126     DD:     DISPLAY DYS
127             RETURN
```

Figure C-2. READTRAN (continued)

```
128
129     ****
130     * A5
131     ****
132     EB:     STORE BATCH.NO
133             COMPARE BATCH.NO NC.ENTRY
134             IF EQUAL NC1
135
136             FIND TIME.REC USING CRT.NO,BATCH.NO
137             ON-IOERR NO.BATCH
138             RETURN
139
140     ****
141     * A6
142     ****
143     DH:     DISPLAY HRS
144             RETURN
145
146     ****
147     * A7
148     ****
149     DMI:    DISPLAY MINUTES
150             RETURN
151
152     ****
153     * A8
154     ****
155     DI:     DISPLAY IDENT
156             RETURN
157
158     ****
159     * A9
160     ****
161     DF:     DISPLAY "TRANSFILE "
162             RETURN USING FIELD
163
164     EF:     STORE FMT.NAME
165             COMPARE FMT.NAME SPACE
166             IF EQUAL NB1
167             VERIFY NO.REC USING CRT.NO BATCH.NO FMT.NAME
168             ON-IOERR NO.FORMAT
169
170     EF1:    MOVE TEN GRP
171             FIND GROUP.HEAD USING CRT.NO BATCH.NO FMT.NAME GRP
172             ON-IOERR NO.GROUP
173             RETURN
174
175     *****
176     * A10
177     *****
178     DG:     DISPLAY GRP
179             RETURN
180
181     *****
182     * A11
183     *****
184     DE:     COMPARE ENTRIES ZERO
185             IF EQUAL NO.TRANS
186
187             DISPLAY ENTRIES
188             RETURN
189
```

Figure C-2. READTRAN (continued)

```
190     *****
191     * A12
192     *****
193     DSFL:   DISPLAY SUM.FL
194             RETURN
195
196     *****
197     * A13
198     *****
199     DTLN:
200             DISPLAY TRAN.LN
201             RETURN
202
203     *****
204     * A14
205     *****
206     DGN:    DISPLAY GROUP.NO
207             RETURN
208
209     *****
210     * A15
211     *****
212     DL:     MOVE SPACE TO PART1
213             MOVE SPACE TO PART2
214             MOVE SPACE TO PART3
215             MOVE SPACE TO PART4
216             MOVE SPACE TO PART5
217             COMPARE FLAG "0"
218             ADD "1" FLAG FLAG.
219             IF NOT-EQUAL GETNEXT
220
221
222             FIND TRAN.REC USING CRT.NO,BATCH.NO,FMT.NAME,GRP,LNNO
223             ON-IOERR NO.TRANS
224
225     DLN1:
226             MOVE TRAN.LN PARTS
227             MOVE ZERO COUNTER
228             RETURN
229     GETNEXT:
230             FIND NEXT TRAN.REC
231             ON-IOERR NO.TRANS
232             GO TO DLN1
233
234
235     *****
236     * A16
237     *****
238     DP1:
239             ADD COUNTER "1" COUNTER
240             GO TO P1,P2,P3,P4,P5 USING COUNTER
241             GO TO END.OF.TRAN
242
243
244     P1:     DISPLAY PART1
245             GO TO TO P6
246
247     P2:     DISPLAY PART2
248             GO TO TO P6
249
250     P3:     DISPLAY PART3
251             GO TO TO P6
252
253     P4:     DISPLAY PART4
254             GO TO TO P6
255
```

Figure C-2. READTRAN (continued)

```
256     P5:     DISPLAY PARTS
257
258     P6:     SUBTRACT "40" PARTS PARTS
259             COMPARE PARTS "1"
260             IF LESS END.OF.TRAN
261             RETURN
262
263     END.OF.TRAN:
264             RETURN A15
265
266     *****
267     * A17
268     *****
269     NEXT:   MOVE "0" FLAG
270             RETURN A10
271
272
273
274     *************
275     * BRANCH CODE
276     *************
277     NO.CRT:
278             MESSAGE NO TRANSACTIONS FROM THIS CRT
279
280     NC1:    RETURN A1
281
282     NO.BATCH:
283             MESSAGE BATCH NOT ENTERED FOR ABOVE CRT
284
285     NB1:    RESET A9
286             RETURN A5
287
288     NO.FORMAT:
289             COMPARE IOERR NOREC
290             IF EQUAL EF1
291             MESSAGE FORMAT NOT ENTERED FOR ABOVE BATCH
292             RETURN USING FIELD
293
294     NO.GROUP:
295             MESSAGE NO TRANSACTIONS FOR ABOVE FORMAT
296             RETURN USING FIELD
297
298     NO.TRANS:
299             ADD GRP TEN GRP
300             FIND GROUP.HEAD USING CRT.NO,BATCH.NO,FMT.NAME,GRP
301             ON=IOERR NT1
302             COMPARE SUM.FL ZERO * SEE NOTE BELOW
303             IF LESS NO.TRANS
304             RETURN A17
305
306     NT1:    MESSAGE END OF TRANSACTIONS ABOVE FORMAT
307             RESET A9
308             RETURN  A1
309
310     * NOTE.  THE FINAL GROUP HEADER IS A DUMMY
311     * RECORD IN WHICH THE FIRST TWO WORDS ARE SET TO
312     * MINUS ONE. SUM.FL IS THE SECOND SUCH WORD.
313     * THE CODE SHOWN SKIPS SUCH RECORDS.
314     *
315             FINISH
```

Figure C-2. READTRAN (continued)

```
 FORMAT NOT LINKED

READTRAN

FIELD      PHYS./LOG.                                        AUTO-  REG.  FULL  AUTO-
NAME         FIELD#   DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

CRT.NO         1    1  9(2)             *                              *
YRS            2    2  9(2)        *
MTH            3    3  9(2)        *
DYS            4    4  9(2)        *
BATCH.NO       5    5  X(3)             *                              *
HRS            6    6  9(2)        *
MINUTES        7    7  9(2)        *
IDENT          8    8  X(10)       *
FMT.NAME       9    9  X(10)       *    *
FILLER        10   10  9(2)        *
ENTRIES       11   11  9(1)        *
SUM.FL        12   12  S9(3)       *
TRAN.LN       13   13  9(3)        *
GROUP.NO      14   14  9(4)        *
FILLER        15   15  X(1)             *                              *
PART1         16   16  X(45)       *
FILLER        17   17  X(1)             *                              *
```

Figure C-2. READTRAN (continued)

You can exercise READTRAN by writing to TRANS with a format such as TRANSFILE, as shown in Figure C-3.

```
12:55:55 03/10/78
PRINT OF FORMAT: TRANSFILE

               **************************************************
               *                  TRANSFILE                     *
               *                  nnnnnnnnn                     *
               * THIS SCREEN WRITES RECORDS TO THE TRANS FILE.*
               *   THERE IS NO PROGRAM ASSOCIATED WITH IT.     *
               **************************************************



          PART 1      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

          PART 2      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

          PART 3      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

          PART 4      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

          PART 5      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


PRINT OF FORMAT: TRANSFILE
     PHYS./LOG.                                          AUTO-   REQ.    FULL    AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE    ENTRY   FIELD   ENTRY   SEC
         1   0  X(40)                    *                                 *
         2   0  X(40)                    *                                 *
         3   0  X(40)                    *                                 *
         4   0  X(40)                    *                                 *
         5   0  X(40)                    *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 20
```

Figure C-3. TRANSFILE

End of Appendix

Appendix D
Format/Program Module Listing

Table D-1. Demonstration Modules

Formats	IFPL Program	Printing Formats	Use
1. DASHJR	DASHJR.UP	PAGEFMT	A simple printing format.
2. DASHDRVR	DASHDRVR.UP	PAGEFMT SCRLLFMT	DASHDRVR creates two print images simultaneously -- the first a transaction-by-transaction copy of the terminal session, the second a scrolled summary of the session's activity. DASHDRVR also writes the transactions to the database, from which point other demonstration programs can read them.
3. DASHCOMM DASHLINK	DASHCOMM.UP DASHLINK.UP		This pair of formats simulates PRINTF by producing DASHER printouts of COMMON print images.
4. BLUEBEARD GRAYBEARD	BLUEBEARD.UP GRAYBEARD.UP		This pair of formats produces DASHER output from the database records written by DASHDRVR. It utilizes the Idea Inactivity Constant to remain on-line when there are no records to print without using significant system resources.
5. DASHDIAG	DASHDIAG.UP		DASHDIAG is a video display used in conjunction with BLUEBEARD. Its job is to reset record flags and the record counter when printing records more than once.
6. HSPA7	HSPA7.UP		This is a format from a hospital system. It demonstrates printing on a satellite DASHER.
7. BIGFOOT	BIGFOOT.UP	PTITLE	This is a format for generating large-character print images.
8. CRAIGS BARGRAPH	CRAIGS.UP BARGRAPH.UP		This pair of formats displays (in bar graph form) data commonly found in company annual reports.

```
13:00:41 02/03/78
PRINT OF FORMAT: DASHJR
X                                INVOICES



 DATE 99/99/99                                              P.O. XXXXXX


**************************
*  XXXXXXXXXXXXXXXXXXX *
*         NAME         *
**************************  ************************
                          * XXXXXXXXXXXXXXXXXXXX *
                          *       ADDRESS        *
                          ************************
                                                    ****************************
                                                    *  XXXXXXXXXXXXXXXXXXXX  *
                                                    *   CITY, STATE, ZIP     *
                                                    ****************************


AGAIN  X


*******************************************************************************
DASHJR  IS  A SIMPLE PRINTING PROGRAM. IT WRITES THE ENTIRE CONTENTS OF
THE  SCREEN AS A SINGLE PAGE RECORD TO THE COMMON PRINT FILE. WHEN THIS
IS  DONE  A  TERMINATE  PRINTING STATEMENT IS EXECUTED, CLOSING OUT THE
PRINT  IMAGE.  WHEN  PRINTING,  THIS RECORD IS ALLOTTED AN ENTIRE FORM.
THIS ARRANGEMENT SATISFIES THE DESIGN INTENT OF PRINTF, WHICH THINKS OF
PRINT IMAGES AS THAT AMOUNT OF DATA THAT WILL FILL ONE FORM.

THE PRINTING FORMAT FOR DASHJR IS "PAGEFMT".
*******************************************************************************
13:00:41 02/03/78
PRINT OF FORMAT: DASHJR
     PHYS./LOG.                                    AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT     DUPE   ENTRY  FIELD  ENTRY  SEC
         1   1  X(1)          *
         2   2  9(2)          *
         3   3  9(2)          *
         4   4  9(2)          *
         5   5  X(6)                *
         6   6  X(20)               *
         7   7  X(20)               *
         8   8  X(20)               *
         9   9  X(1)                *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 23
AOS SYNTAX REV 01.01     DASHJR.VS   DASHJR.UP  13:1:5   2/3/78
```

Figure D-1. DASHJR

```
1
2
3
4       NAME DASHJR
5       *DASHJR IS AN EXAMPLE OF A SIMPLE IFPL PRINTING PROGRAM
6       *PLEASE SEE NOTES AT THE END FOR AN EXPLANATION.
7
8       RECORD FOR PRINTING IS PAGEREC
9            LENGTH IS 78
10           INCLUDES INVNO 1 6 ASCII
11           INCLUDES MONTH 7 2 ASCII
12           INCLUDES DAY 9 2 ASCII
13           INCLUDES YEAR 11 2 ASCII
14           INCLUDES PO 13 6 ASCII
15           INCLUDES NAME 19 20 ASCII
16           INCLUDES ADDRESS 39 20 ASCII
17           INCLUDES CITY 59 20 ASCII
18           STOP
19
20      REGISTER INVNO 9(6) 0
21
22
23
24
25              PROCESS FILLER AT D1 AND NONE
26              PROCESS MONTH AT D2 AND NONE
27              PROCESS DAY AT D3 AND NONE
28              PROCESS YEAR AT D4 AND NONE
29              PROCESS PO AT NONE AND E5
30              PROCESS NAME AT NONE AND E6
31              PROCESS ADDRESS AT NONE AND E7
32              PROCESS CITY AT NONE AND E8
33              PROCESS FILLER AT NONE AND E9
34
35
36      D1:
37                  INITIATE PRINTING USING "PAGEFMT"
38                  RETURN
39
40      D2:
41                  DISPLAY MONTH
42                  RETURN
43
44      D3:
45                  DISPLAY DAY
46                  RETURN
47
48      D4:
49                  DISPLAY YEAR
50                  RETURN
51
52      E5:
53                  STORE PO
54                  RETURN
55
56      E6:
57                  STORE NAME
58                  RETURN
59
60      E7:
61                  STORE ADDRESS
62                  RETURN
63
```

Figure D-1. DASHJR (continued)

```
64      E8:
65                  STORE CITY
66                  PRINT PAGEREC USING "PAGEFMT"
67                  TERMINATE PRINTING USING "PAGEFMT"
68                  MESSAGE ONE PAGE GROUP (=1 PRINT IMAGE) WRITTEN TO COMMON
69                  RETURN
70
71
72      E9:
73                  RETURN 1
74
75
76
77      *DASHJR IS DESIGNED TO SATISFY THE REQUIREMENTS OF THE IFPL
78      *PRINTING FACILITY IN THE SIMPLEST POSSIBLE WAY. THE PROGRAM
79      *WRITES A SINGLE PAGE-FORMAT RECORD TO COMMON. IT THEN TERMINATES
80      *PRINTING, MAKING THE RECORD AND THE PRINT IMAGE COTERMINOUS.
81      *THAT IS, EACH PRINT IMAGE CONTAINS EXACTLY ONE RECORD. FUTHERMORE,
82      *ON PRINTOUT, EACH RECORD FILLS A PAGE.  THIS SATISFIES THE DESIGN
83      *INTENT OF THE PRINT FACILITY, WHICH IS THAT A PRINT IMAGE SHOULD
84      *FILL EXACTLY ONE FORM.
85      *
86      *THE RECORDS THUS WRITTEN TO COMMON CAN BE PRINTED OUT TWO TO A FORM.
87      *THIS IS DONE BY CREATING A PRINTING FORMAT 31 LINES LONG; BY LINKING
88      *IT TO ITSELF; AND BY SUPPRESSING THE FORM FEED ON LINKING.
89      *
90      *WHEN CREATING THE PRINTING FORMAT, THE DEFAULT IS TAKEN ON THE FORM
91      *LENGTH, MAKING IT 66 LINES LONG.  IF THE LENGTH IS SET AT ANY
92      *THING LESS ON A 66-LINE PRINTER -- SAY, 33 LINES -- PRINTF
93      *WILL ISSUE A FORM FEED TO THE PRINTER WHEN 33 LINES HAVE BEEN
94      *REACHED.  THE PRINTER WILL RESPOND BY GOING TO THE TOP OF THE NEXT
95      *FORM, AND THE LOWER HALF OF THE FORM WILL BE LEFT BLANK.
96      *FORM LENGTHS THUS CANNOT BE LESS THAN THAT OF THE PRINTER BEING
97      *USED.  THEY CAN, HOWEVER, BE MORE.
98      *
99      *31 LINES EQUALS (66-4)/2
100     *
101     *TWO-UP PRINTING CANNOT BE DONE BY REPEATING THE FORMAT ON THE
102     *LOWER HALF OF THE FORM -- THAT IS, BY ASKING PRINTF
103     *TO WRITE ONE RECORD ON THE TOP HALF AND A SECOND ON THE
104     *BOTTOM HALF. THIS VIOLATES ITS RULE OF HAVING THE DATA AND THE
105     *FORMAT END SYNCHRONOUSLY.
106     *
107                 FINISH
 FORMAT NOT LINKED

DASHJR

FIELD     PHYS./LOG.                                    AUTO-  REG.  FULL  AUTO-
NAME        FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

FILLER       1   1  X(1)          *
MONTH        2   2  9(2)          *
DAY          3   3  9(2)          *
YEAR         4   4  9(2)          *
PC           5   5  X(6)               *
NAME         6   6  X(20)              *
ADDRESS      7   7  X(20)              *
CITY         8   8  X(20)              *
FILLER       9   9  X(1)               *
14:39:43 02/02/78
PRINT OF FORMAT: PAGEFMT
```

Figure D-1. DASHJR (continued)

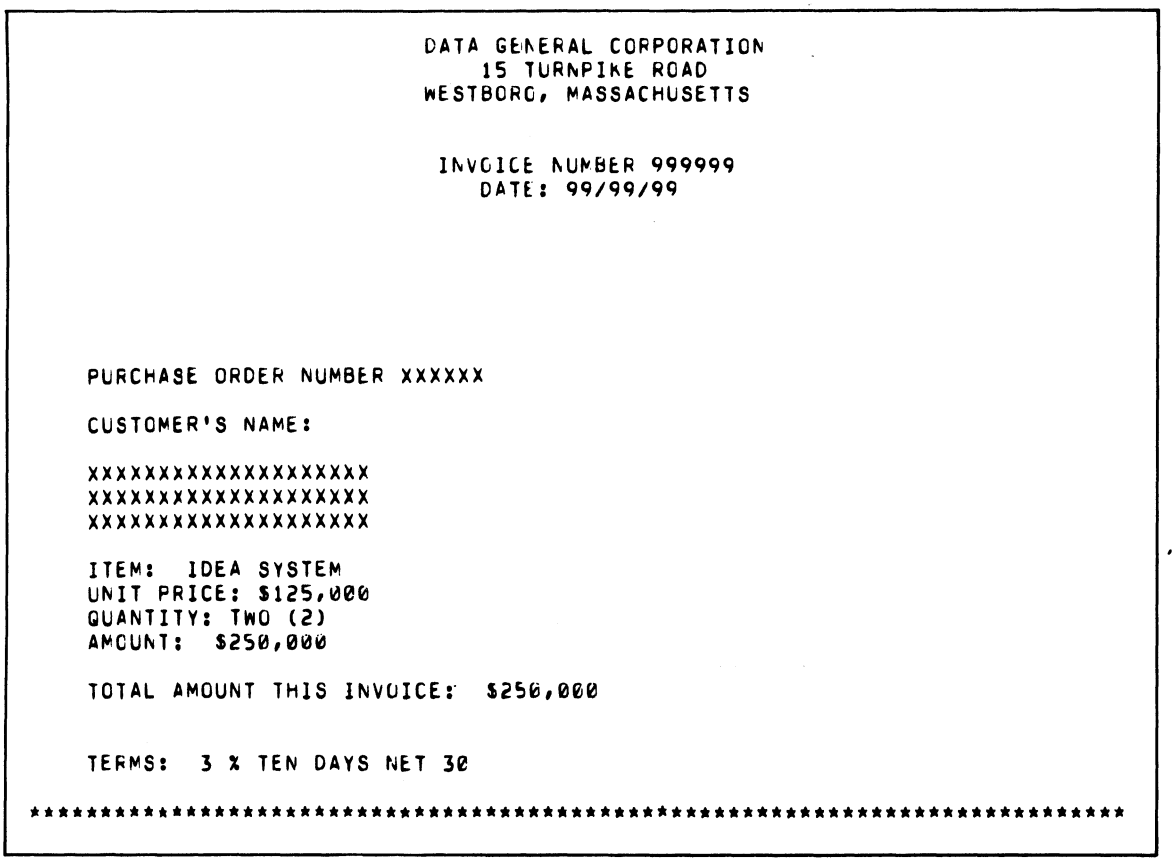


Figure D-1. DASHJR (continued)

```
14:39:43 02/02/78
PRINT OF FORMAT: PAGEFMT
     PHYS./LOG.                                        AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
        1    0  9(6)
        2    0  9(2)
        3    0  9(2)
        4    0  9(2)
        5    0  X(6)
        6    0  X(20)
        7    0  X(20)
        8    0  X(20)
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 31
13:03:57 02/03/78
PRINT OF FORMAT: DASHDRVR
X                                INVOICES



 DATE 99/99/99                                                P.O. XXXXXX


**************************
*  XXXXXXXXXXXXXXXXXXX *
*          NAME        *
**************************  *************************
                            * XXXXXXXXXXXXXXXXXXXX *
                            *       ADDRESS       *
                            *************************
                                                     ****************************
                                                     *  XXXXXXXXXXXXXXXXXXXX  *
                                                     *    CITY, STATE, ZIP    *
                                                     ****************************


AGAIN  X



*******************************************************************************
DASHDRVR IS THE ADULT VERSION OF DASHJR. THE SCREEN FORMAT FOR THE
TWO MODULES IS THE SAME. BUT DASHDRVR CREATES TWO PRINT IMAGES
SIMULTANEOUSLY -- THE SIMPLE SCREEN IMAGE OF DASHJR, PLUS A SCROLLED
SUMMARY OF THE ENTIRE TERMINAL SESSION.

IN ADDITION, DASHDRVR WRITES A RECORD REFLECTING EACH TRANSACTION
TO THE DATABASE FILE INVOICES.

THE PRINT IMAGES OF DASHDRVR ARE READ OUT WITH THE PRINTING
FORMATS "PAGEFMT" AND "SCRLLFMT".
*******************************************************************************
```

Figure D-2. DASHDRVR

```
13:03:57 02/03/78
PRINT OF FORMAT: DASHDRVR
     PHYS./LOG.                                          AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
        1   1  X(1)          *
        2   2  9(2)          *
        3   3  9(2)          *
        4   4  9(2)          *
        5   5  X(6)                *
        6   6  X(20)               *
        7   7  X(20)               *
        8   8  X(20)               *
        9   9  X(1)                *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 23
AOS SYNTAX REV 01.01     DASHDRVR.VS   DASHDRVR.UP  13:4:28   2/3/78



1
2
3
4       NAME DASHDRVR
5       *DASHDRVR IS THE ADULT VERSION OF THE PRINTING PROGRAM, DASHJR.
6       *IT PRINTS TWO TYPES OF REPORTS -- ONE CONSISTING OF
7       *ALL THE INVOICES ENTERED INTO THE DATABASE DURING THE DAY,
8       *THE OTHER CONSISTING OF A CONDENSED SUMMARY REPORT OF THE DAY'S
9       *ACTIVITIES. THESE ARE KEYED BY "PAGEFMT" AND "SCRLLFMT" RESPECTIVELY.
10      *
11      *DASHDRVR IS ALSO THE DRIVING PROGRAM FOR A SET OF PROGRAMS THAT
12      *UTILIZE THE DASHER FOR PRINTING.  THESE INCLUDE
13      *
14      *          BLUEBEARD
15      *          DASHDIAG
16      *
17      *BLUEBEARD OUTPUTS DATABASE RECORDS ON THE DASHER TERMINAL, SETTING
18      *A FLAG ON THE RECORD TO INDICATE IT HAS BEEN PRINTED.  THIS FLAG
19      *CAN BE RESET WITH THE CRT TERMINAL PROGRAM DASHDIAG.
20      *
21      *FURTHER DETAILS ON PRINTING WITH DASHDRVR ARE GIVEN BELOW ADJACENT
22      *TO THE FINISH STATEMENT.
23      *
24      FILE INVOICES
25      KEY FOR INVOICES IS 6 ASCII
26      RECORD FOR INVOICES IS INVREC
27           LENGTH IS 79
28           INCLUDES INVNO 1 6 ASCII
29           INCLUDES MONTH 7 2 ASCII
30           INCLUDES DAY 9 2 ASCII
31           INCLUDES YEAR 11 2 ASCII
32           INCLUDES PO 13 6 ASCII
33           INCLUDES NAME 19 20 ASCII
34           INCLUDES ADDRESS 39 20 ASCII
35           INCLUDES CITY 59 20 ASCII
36           INCLUDES PRTFLG 79 1 ASCII
37           STOP
38
39
```

Figure D-2. DASHDRVR (continued)

```
40      RECORD FOR PRINTING IS PAGEREC
41           LENGTH IS 78
42           INCLUDES INVNO 1 6 ASCII
43           INCLUDES MONTH 7 2 ASCII
44           INCLUDES DAY 9 2 ASCII
45           INCLUDES YEAR 11 2 ASCII
46           INCLUDES PO 13 6 ASCII
47           INCLUDES NAME 19 20 ASCII
48           INCLUDES ADDRESS 39 20 ASCII
49           INCLUDES CITY 59 20 ASCII
50           STOP
51
52
53      RECORD FOR PRINTING IS DATEREC
54           LENGTH IS 8
55           INCLUDES DATE 1 8 ASCII
56           STOP
57
58      RECORD FOR PRINTING IS SCROLLREC
59           LENGTH IS 66
60           INCLUDES INVNO 1 6 ASCII
61           INCLUDES NAME 7 20 ASCII
62           INCLUDES ADDRESS 27 20 ASCII
63           INCLUDES CITY 47 20 ASCII
64           STOP
65
66
67      RECORD FOR PRINTING IS ENDSCROLL
68           LENGTH IS 1
69           INCLUDES "@" 1 1 ASCII
70           STOP
71
72      REGISTER PRTFLG 9(1) 0
73      REGISTER DATE  X(8)  00/00/00
74      REDESIGNATE DATE
75           MONTH  1  2
76           DAY    4  2
77           YEAR   7  2
78           STOP
79      REGISTER KEYKOUNT 9(2) 0
80      REGISTER INVNO 9(6) 0
81
82
83      ON END OF DATA END
84      ON LOGOFF END
85
86
87              PROCESS FILLER AT D1 AND NONE
88      A1#     PROCESS FILLER AT D2 AND NONE
89              PROCESS FILLER AT D3 AND NONE
90              PROCESS FILLER AT D4 AND NONE
91              PROCESS PO AT NONE AND E5
92              PROCESS NAME AT NONE AND E6
93              PROCESS ADDRESS AT NONE AND E7
94              PROCESS CITY AT NONE AND E8
95              PROCESS FILLER AT NONE AND E9
96
97
98      D1:
99                  INITIATE PRINTING USING "SCRLLFMT"
100                 INITIATE PRINTING USING "PAGEFMT"
101                 PRINT DATEREC USING "SCRLLFMT"
102                 RETURN
103
104
```

Figure D-2. DASHDRVR (continued)

```
105     D2:
106                 DISPLAY MONTH
107                 RETURN
108
109     D3:
110                 DISPLAY DAY
111                 RETURN
112
113     D4:
114                 DISPLAY YEAR
115                 RETURN
116
117     E5:
118                 STORE PO
119                 RETURN
120
121     E6:
122                 STORE NAME
123                 RETURN
124
125     E7:
126                 STORE ADDRESS
127                 RETURN
128
129     E8:
130                 STORE CITY
131                 VERIFY INVREC USING "000001"
132                 ON-IOERR E8D
133     E8A:
134                 RETRIEVE HIGH KEY FOR INVREC TO INVNO
135                 ADD "1" INVNO INVNO
136                 FILE-NEW INVREC USING INVNO
137                 ON-IOERR E8C
138     E8B:
139                 PRINT PAGEREC USING "PAGEFMT"
140                 PRINT SCROLLREC USING "SCRLLFMT"
141                 MESSAGE ONE PAGE GROUP AND ONE SCROLL LINE WRITTEN TO COMMON
142                 RETURN
143
144     E8C:
145                 ADD "1" KEYKOUNT KEYKOUNT
146                 COMPARE KEYKOUNT "10"
147                 IF GREATER E8G
148                 GO TO E8A
149
150     E8D:
151                 MOVE "1" INVNO
152                 FILE-NEW INVREC USING INVNO
153                 ON-IOERR E8E
154                 GO TO E8B
155
156     E8E:
157                 MESSAGE FATAL WRITE ERROR ON INITIAL RECORD
158                 QUIT
159
160     E8G:
161                 MESSAGE FATAL WRITE ERROR.
162                 QUIT
163
164     E9:
165                 RETURN A1
166
167
```

Figure D-2. DASHDRVR (continued)

```
168     END:
169                 TERMINATE PRINTING USING "PAGEFMT"
170                 PRINT ENDSCROLL USING "SCRLLFMT"
171                 TERMINATE PRINTING USING "SCRLLFMT"
172                 MESSAGE PRINTING USING "PAGEFMT" AND "SCRLLFMT" TERMINATED. ↑
173     PROGRAM LOGGED OFF.
174
175                 QUIT
176
177     *
178     *PRINTF HAS A DEFAULT FOR AN ERROR CONDITION THAT CONSISTS OF HAVING MORE
179     *DATA THAN WILL FIT THE PRINTING FORMAT. THE DEFAULT IS THAT IT DOES A
180     *FORM FEED AND A RESTART OF THE FORMAT. THIS DEFAULT HAS BEEN UTILIZED IN
181     *THIS PROGRAM TO BUILD TWO REPORTS SIMULTANEOUSLY.
182     *
183     *DASHDRVR UTILIZES A SINGLE PRINT IMAGE PER KEYBOARD SESSION FOR
184     *EACH OF THE TWO PRINTING FORMATS IT DRIVES.  IT DOES THIS BY INITIATING
185     *PRINTING IN A DUMMY FIELD AT LOG ON, AND NEVER RETURNING TO THAT FIELD.
186     *TERMINATE PRINTING STATEMENTS ARE EXECUTED ONCE ONLY, AT LOG OFF.
187     *
188     *COMMON USES A SINGLE RECORD COUNTER FOR BOTH PRINT FORMATS BEING
189     *WRITTEN TO. IN THE DASHDRVR CODE,A SINGLE SCROLL FORMAT RECORD
190     *IS WRITTEN AT LOG ON.  THEREAFTER, THE PROGRAM ALTERNATELY WRITES
191     *RECORDS TO EACH OF THE TWO PRINTING FORMATS, STARTING WITH THE PAGE FORMAT.
192     *A KEYBOARD SESSION THUS WRITES TO COMMON SCROLL RECORDS WHOSE BOTTOM KEYS
193     *ARE 1,3,5 . . . AND WHOSE BOTTOM PAGE KEYS ARE 2,4,6 . . .  THIS IS
194     *NOT CONFUSING TO PRINTF, WHICH ONLY REQUIRES THAT THE NUMBERS FOR
195     *EACH SET BE IN ASCENDING ORDER, BUT THE PAGE RECORDS COULD NOT BE
196     *TERMINATED ASYNCHRONOUSLY WITH THE SCROLL RECORDS.  THAT IS, THE SCROLL
197     *IMAGE CANNOT BE LEFT OPEN AFTER THE PAGE IMAGE HAS BEEN CLOSED. WERE THIS
198     *TO HAPPEN, THE NEXT SCROLL RECORD WRITTEN WOULD HAVE A DUPLICATE KEY
199     *ERROR, SINCE TERMINATING THE PAGE IMAGE RESETS THE RECORD COUNTER WHICH
200     *SUPPLIES KEYS TO BOTH IMAGES.
201     *
202     *ANOTHER REQUIREMENT FOR BUILDING DUAL IMAGES AS ABOVE IS THAT IT MUST
203     *ALL BE DONE WITHOUT LEAVING THE PROGRAM.  THAT IS, RETURNS MUST BE BY
204     *RETURN STATEMENTS, NOT BY LINKING.  THE LATTER INCREMENTS THE DUPLICATES
205     *COUNT AND RESETS THE RECORD COUNTER.  NEITHER ACTION IS WELCOME HERE.
206     *WHEN PRINTING OUT, PRINTF PERFORMS AS THOUGH EACH PAGE RECORD HAD BEEN
207     *ASSOCIATED WITH ITS OWN EXCLUSIVE PRINT IMAGE.
208     *
209     *THE PRINTING FORMAT FOR THE SCROLL RECORDS, "SCRLLFMT", USES REPEATED
210     *HEADINGS, BOTH PAGE AND SCROLL. PRINTF ALLOWS ONE OF EACH TYPE.
211     *THE FORMAT MUST, HOWEVER, BE INITIATED IN PAGE MODE FOR THIS. THIS
212     *IS ACCOMPLISHED BY PRINTING A SINGLE FIELD OF PAGE DATA ON THE FIRST
213     *FORM OF THE SCROLL SERIES. WITHOUT THIS FIELD, THE FORMAT WOULD BE
214     *ENTERED IN SCROLL MODE; ALL REPEATED LITERALS WOULD BE TREATED AS
215     *SCROLL HEADINGS; AND ONLY THE LAST GIVEN WOULD BE PRINTED.
216     *
217     *THE PAGE RECORDS WRITTEN TO COMMON WITH DASHDRVR ARE PRINTED WITH THE
218     *IDENTICAL FORMAT USED FOR DASHJR. HERE, HOWEVER, THEY PRINT OUT
219     *ONE TO A FORM.  THIS IS BECAUSE, WHEN THE FORMAT IS SATISFIED,
220     *THERE IS DATA LEFT.  PRINTF THEN TAKES THE DEFAULT MENTIONED ABOVE,
221     *AND ISSUES A FORM FEED TO THE LINE PRINTER.
222     *
223     *THE SCROLL FORMAT IS LINKED TO THE PAGE FORMAT.  IF PRINTF IS TOLD
224     *TO PRINT OUT THE SCROLL RECORDS, IT WILL DO SO, AND THEN PRINT
225     *OUT ALL THE PAGE RECORDS WHEN IT LINKS.
226     *
227                 FINISH
```

Figure D-2. DASHDRVR (continued)

```
 FORMAT NOT LINKED

DASHDRVR

FIELD     PHYS./LOG.                                    AUTO-  REQ.  FULL  AUTO-
NAME        FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

FILLER        1   1  X(1)          *
FILLER        2   2  9(2)          *
FILLER        3   3  9(2)          *
FILLER        4   4  9(2)          *
PC            5   5  X(6)               *
NAME          6   6  X(20)              *
ADDRESS       7   7  X(20)              *
CITY          8   8  X(20)              *
FILLER        9   9  X(1)               *
14:40:07 02/02/78
PRINT OF FORMAT: SCRLLFMT
```

Figure D-2. DASHDRVR (continued)

```
                        DATA GENERAL CORPORATION

                        //DGC DAILY INVOICE RECORD//

                                 XXXXXXXX

      COPIES: ACCOUNTING, PURCHASING, LEGAL, MANUFACTURING, MARKETING, FILE


@//INVOICE NO.   CUSTOMER'S NAME              ADDRESS          CITY, STATE, ZIP//

  999999     XXXXXXXXXXXXXXXXXXX     XXXXXXXXXXXXXXXXXXX    XXXXXXXXXXXXXXXXXXX
@
14:40:07 02/02/78
PRINT OF FORMAT: SCRLLFMT
     PHYS./LOG.                                    AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT     DUPE   ENTRY  FIELD  ENTRY  SEC
         1   0  X(8)
         2   0  9(6)
         3   0  X(20)
         4   0  X(20)
         5   0  X(20)
LINKED TO FORMAT: PAGEFMT
FIRST LINE USED: 1
LAST LINE USED: 27
14:39:43 02/02/78
PRINT OF FORMAT: PAGEFMT
```

Figure D-2. DASHDR VR (continued)

```
                        DATA GENERAL CORPORATION
                            15 TURNPIKE ROAD
                        WESTBORO, MASSACHUSETTS


                          INVOICE NUMBER 999999
                             DATE: 99/99/99


PURCHASE ORDER NUMBER XXXXXX

CUSTOMER'S NAME:

XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX

ITEM:  IDEA SYSTEM
UNIT PRICE: $125,000
QUANTITY: TWO (2)
AMOUNT:  $250,000

TOTAL AMOUNT THIS INVOICE:  $250,000


TERMS:  3 % TEN DAYS NET 30

********************************************************************************
14:39:43 02/02/78
PRINT OF FORMAT: PAGEFMT
     PHYS./LOG.                                        AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
         1   0  9(6)
         2   0  9(2)
         3   0  9(2)
         4   0  9(2)
         5   0  X(6)
         6   0  X(20)
         7   0  X(20)
         8   0  X(20)
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 31
```

Figure D-2. DASHDRVR (continued)

```
13:07:23 02/03/78
PRINT OF FORMAT: DASHCOMM
X



********************************************************************************
********************************************************************************
13:07:23 02/03/78
PRINT OF FORMAT: DASHCOMM
     PHYS./LOG.                                    AUTO-  REG.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT     DUPE   ENTRY  FIELD  ENTRY  SEC
        1   1  X(1)          *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 1
ACS SYNTAX REV 01.01      DASHCOMM.VS    DASHCOMM.UP  13:7:49   2/3/78



1
2
3
4       NAME DASHCOMM
5       *
6       *THE PURPOSE OF DASHCOMM AND ITS COMPANION PROGRAM, "DASHLINK,"
7       *IS TO SIMULATE THE ACTION OF PRINTF -- THAT IS, TO OUTPUT PRINT
8       *RECORDS FROM THE COMMON FILE TO A PRINTER.
9       *DASHCOMM AND DASHLINK RUN ON THE  DGC DASHER TERMINAL.
10      *THEY READ THE PRINT RECORDS WRITTEN TO THE COMMON FILE
11      *BY THE PROGRAMS "PAGEFMT" AND  "DASHJR," AND PRODUCE A HARDCOPY PRINTOUT.
12      *
13      *THESE PROGRAMS ARE NOT AS SOPHISTICATED AS PRINTF. THEY DISREGARD THE
14      *PRINT FLAG, NEITHER READING NOR WRITING IT. NOR CAN THEY DELETE
15      *RECORDS AS THEY ARE PRINTED.
16      *
17      *THE PURPOSE OF DASHCOMM IS TO INITIALIZE PASSING VARIABLES USED
18      *BY DASHLINK.
19      *
20      RECORD FOR PASSING IS PASSREC
21      LENGTH IS 10
22           INCLUDES CRTNO  1  2  ASCII
23           INCLUDES DUPES  3  4  ASCII
24           INCLUDES RECNO  7  4  ASCII
25           STOP
26      REGISTER CRTNO     9(2)  0
27      REGISTER DUPES     9(4)  0
28      REGISTER RECNO     9(4)  1
29      REGISTER L1        X(8)  DASHLINK
30      PROCESS FILLER AT D1 AND NONE
31
32      D1:
33                 PASS PASSREC
34                 LINK USING L1
35
36      FINISH
```

Figure D-3. DASHCOMM

```
 FORMAT NOT LINKED

DASHCOMM

FIELD      PHYS./LOG.                                    AUTO- REQ.  FULL  AUTO-
NAME         FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

FILLER         1   1  X(1)          *
13:10:03 02/03/78
PRINT OF FORMAT: DASHLINK
ZZZZZZ
99/99/99
PO XXXXXX
NAME     XXXXXXXXXXXXXXXXXXX
ADDRESS XXXXXXXXXXXXXXXXXXX
CITY     XXXXXXXXXXXXXXXXXXX


   X X


**********************************************************************************
```

Figure D-3. DASHCOMM (continued)

```
********************************************************************************
13:10:03 02/03/78
PRINT OF FORMAT: DASHLINK
     PHYS./LOG.                                        AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
        1   1  9(6)          *
        2   2  9(2)          *
        3   3  9(2)          *
        4   4  9(2)          *
        5   5  X(6)          *
        6   6  X(20)         *
        7   7  X(20)         *
        8   8  X(20)         *
        9   9  X(1)          *
        10  10 X(1)          *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 65
ACS SYNTAX REV 01.01     DASHLINK.VS   DASHLINK.UP  13:10:39   2/3/78


1
2
3
4       NAME DASHLINK
5       *
6       *THE PURPOSE OF THIS PROGRAM IS TO PRODUCE A PRINTED COPY
7       *OF THE "PAGEFMT"  PRINT RECORDS WRITTEN TO COMMON BY THE PROGRAMS
8       *"DASHJR" AND "DASHDRVR."   IT IS RUN ON A DGC DASHER TERMINAL.
9       *IT IS NOT ENTERED DIRECTLY, BUT THROUGH "DASHCOMM,"
10      *WHICH SERVES TO INITIALIZE PASSING VARIABLES.
11      *
12      *
13      *DASHLINK LINKS TO ITSELF TO CONTINUE, DOING SO ON SUCH A
14      *LINE THAT IT SIMULATES A FORM FEED.
15      *
16      FILES COMMON
17      KEY FOR COMMON IS 13 ASCII
18      SUBINDEX FOR COMMON IS LEVEL1
19      KEY FOR LEVEL1 IS 2 ASCII
20      DUPLICATES ARE COUNTED IN DUPES
21      RECORD FOR LEVEL1 IS LEVEL1REC
22           LENGTH IS 2
23           INCLUDES IMAGES 1 2 BINARY
24           STOP
25      SUBINDEX FOR LEVEL1 IS LEVEL2
26      KEY FOR LEVEL2 IS 2 BINARY
27      RECORD FOR LEVEL2 IS PRINTREC
28           LENGTH IS 78
29           INCLUDES INVNO      1  6  ASCII
30           INCLUDES IMONTH     7  2  ASCII
31           INCLUDES IDAY       9  2  ASCII
32           INCLUDES IYEAR     11  2  ASCII
33           INCLUDES PO        13  6  ASCII
34           INCLUDES NAME      19 20  ASCII
35           INCLUDES ADDRESS   39 20  ASCII
36           INCLUDES CITY      59 20  ASCII
37           STOP
38      RECORD FOR PASSING IS PASSREC
39      LENGTH IS 10
40           INCLUDES CRTNO  1  2  ASCII
41           INCLUDES DUPES  3  4  ASCII
42           INCLUDES RECNO  7  4  ASCII
43           STOP
44
```

Figure D-3. DASHCOMM (continued)

```
45      REGISTER DUPES      9(4)
46      REGISTER CRTNO      9(2)
47      REGISTER RECNO      9(4)
48      REGISTER L1          X(8)  DASHLINK
49      REGISTER IMAGES     9(4)
50
51
52
53      PROCESS INVNO     AT D1  AND NONE
54      PROCESS IMONTH     AT D2  AND NONE
55      PROCESS IDAY       AT D3  AND NONE
56      PROCESS IYEAR      AT D4  AND NONE
57      PROCESS PO        AT D5  AND NONE
58      PROCESS NAME      AT D6  AND NONE
59      PROCESS ADDRESS   AT D7  AND NONE
60      PROCESS CITY      AT D8  AND NONE
61      PROCESS FILLER    AT D9  AND NONE
62      PROCESS FILLER    AT D10 AND NONE
63
64      *THIS PROGRAM LOOKS FOR A PRINTREC USING AS THE LEVEL 0 KEY THE FORMAT
65      *NAME "PAGEFMT".  IT LOOKS FIRST FOR A RECORD WITH KEYS "PAGEFMT",
66      *"00", "0001", DUPLICATE COUNT = 0.  THE LEVEL1 KEY IS THE CRT NUMBER.
67      *THE LEVEL2 KEY IS THE PRINT RECORD NUMBER.
68
69      *AFTER THE PRINT RECORDS ARE EXHAUSTED WITH THE ABOVE KEYS, THE
70      *DUPLICATES COUNT IS INCREMENTED BY "1" AND ALL PRINT RECORDS
71      *ASSOCIATED WITH THE NEW KEY SPECIFICATIONS ARE PRINTED.
72
73      *WHEN THERE ARE NO MORE DUPLICATES, THE CRT NUMBER IS INCREMENTED
74      *BY A FIND NEXT STATEMENT LOOKING FOR THE NEXT PRINT FLAG
75      *(LEVEL1REC). WHEN THERE ARE NO MORE PRINTFLAGS, THE PROGRAM
76      *TERMINATES.
77
78      *FORM FEEDS ARE SIMULATED BY EXECUTING A DUMMY FIELD ON LINE
79      *66 OF THE FORMAT.
80
81
82      D1:
83                  ACCEPT PASSREC
84      D1FIND:
85                  FIND THE PRINTREC NEAREST "PAGEFMT",CRTNO,RECNO
86                  ON-IOERR D1A  *NO MORE RECNO'S. TRY
87                                *ANOTHER DUPLICATE.
88              RETRIEVE KEY FOR PRINTREC TO RECNO
89                  DISPLAY INVNO
90                  RETURN
91
92      *THIS CODE DOESN'T WORK. DUPES DOESN'T SEEM TO BE UPDATED.
93      *D1A:
94              FIND THE LEVEL1REC NEAREST "PAGEFMT",CRTNO
95              ON-IOERR D1C
96              COMPARE RECNO "0"
97              IF EQUAL D1B
98              FIND THE NEXT LEVEL1REC
99              ON-IOERR D1C
100
101     *D1B:
102             RETRIEVE KEY FOR LEVEL1REC TO CRTNO
103             MOVE "0" RECNO
104             GO TO D1FIND
105
```

Figure D-3. DASHCOMM (continued)

```
106     D1A:
107                 ADD "1" DUPES DUPES
108                 FIND LEVEL1REC USING "PAGEFMT", CRTNO
109                 ON-IOERR D1B  *NO MORE PRINTRECS. TRY
110                               *ANOTHER CRT.
111                 MOVE "1" RECNO
112                 GO TO D1FIND
113
114     D1B:
115             MOVE "0" DUPES
116             ADD "1" CRTNO CRTNO
117             FIND THE LEVEL1REC NEAREST "PAGEFMT",CRTNO
118             ON-IOERR D1C
119             RETRIEVE KEY FOR LEVEL1REC TO CRTNO
120     *DUPES IS NOW SET TO "0"
121                 MOVE "1" RECNO
122                 GO TO D1FIND
123
124     D1C:
125                 RETURN 10
126
127     D2:
128                 DISPLAY IMONTH
129                 RETURN
130
131     D3:
132                 DISPLAY IDAY
133                 RETURN
134
135     D4:
136                 DISPLAY IYEAR
137                 RETURN
138
139     D5:
140                 DISPLAY PO
141                 RETURN
142
143     D6:
144                 DISPLAY NAME
145                 RETURN
146
147     D7:
148                 DISPLAY ADDRESS
149                 RETURN
150
151     D8:
152                 DISPLAY CITY
153                 RETURN
154
155     D9:
156                 ADD "1" RECNO RECNO
157                 PASS PASSREC
158                 LINK USING L1
159
160     D10:
161                 MESSAGE NO MORE RECORDS
162                 QUIT
163
164     FINISH
 FORMAT NOT LINKED

DASHLINK
```

Figure D-3. DASHCOMM (continued)

```
FIELD     PHYS./LOG.                                    AUTO- REG.  FULL  AUTO-
NAME        FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

INVNO         1   1  9(6)          *
IMONTH        2   2  9(2)          *
IDAY          3   3  9(2)          *
IYEAR         4   4  9(2)          *
PO            5   5  X(6)          *
NAME          6   6  X(20)         *
ADDRESS       7   7  X(20)         *
CITY          8   8  X(20)         *
FILLER        9   9  X(1)          *
FILLER       10  10  X(1)          *
```

Figure D-3. DASHCOMM (continued)

```
13:13:19 02/03/78
PRINT OF FORMAT: BLUEBEARD

X
********************************************************************************
********************************************************************************
13:13:19 02/03/78
PRINT OF FORMAT: BLUEBEARD
     PHYS./LOG.                                    AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
         1   1  X(1)          *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 61
AOS SYNTAX REV 01.01      BLUEBEARD.VS    BLUEBEARD.UP  13:13:49   2/3/78


1                NAME BLUEBEARD
2        *
3        *BLUEBEARD AND ITS COMPANION PROGRAM, "GRAYBEARD,"
4        *ARE DASHER PRINTING PROGRAMS THAT PRINT WHEN THEY ARE
5        *NEEDED, AND REMAIN INACTIVE BUT ALERT WHEN NOT.  WHILE
6        *INACTIVE, THEY SCAN THE DATA BASE LOOKING FOR RECORDS
7        *TO PRINT. THE ADDITION OF A RECORD TO THE DATA BASE
8        *SIGNALS THEM TO RESUME WORK.
9        *
10       *THEY ARE NAMED AFTER THE LEGENDARY GERMAN HERO, BLUEBEARD,
11       *WHO SLEPT IN HIS MOUNTAIN FASTNESS FOR A HUNDRED YEARS AT
12       *A TIME. HE THEN SALLIED FORTH TO SEE IF HIS COUNTRY
13       *NEEDED HIM. IF SO, HE HELPED OUT.  OTHERWISE HE WENT BACK TO
14       *SLEEP.
15       *
16       *BLUEBEARD THE DASHER PROGRAM DEPENDS ON THE INACTIVITY
17       *CONSTANT OF THE IDEA SYSTEM, WHICH IS SET NOT BY LEGEND BUT BY THE
18       *WRITER OF THE IFPL PROGRAM. ITS UNITS ARE MINUTES RATHER THAN YEARS.
19       *
20       *THE TWO PROGRAMS -- BLUEBEARD AND GRAYBEARD -- PRINT
21       *OUT ON THE DGC DASHER TERMINAL THE DATA BASE RECORDS
22       *WRITTEN TO THE FILE, "INVOICES," BY THE PROGRAM,
23       *"DASHDRVR."  SEE THE LISTING OF GRAYBEARD FOR DETAILS.
24       *
25       *BLUEBEARD SERVES TO SIMULATE AN INITIAL FORM FEED
26       *TO ALIGN THE PRINTHEAD PRIOR TO PRINTING.  IT ALSO
27       *INITIALIZES THE RECORD FOR PASSING IF THIS HAS NOT
28       *ALREADY BEEN DONE.
29       *
30               RECORD FOR PASSING IS POINTEREC
31               LENGTH IS 15
32                       INCLUDES POINTER  1  6  ASCII
33                       INCLUDES SIGNATURE  7  9  ASCII
34                       STOP
35               REGISTER SIGNATURE  X(9)
36               REGISTER POINTER 9(6)
37               REGISTER L1 X(9) GRAYBEARD
38               PROCESS FILLER AT D1 AND NONE
39
40       D1:
41               ACCEPT POINTEREC
42               COMPARE SIGNATURE "SIGNATURE"
43               IF EQUAL D1A
44               MOVE "SIGNATURE" SIGNATURE
45               MOVE "1" POINTER
46               PASS POINTEREC
47
```

Figure D-4. BLUEBEARD and GRAYBEARD

```
48      D1A:
49              LINK USING L1
50
51              FINISH
 FORMAT NOT LINKED

BLUEBEARD

FIELD     PHYS./LOG.                                    AUTO- REG.  FULL  AUTO-
NAME        FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

FILLER       1   1  X(1)          *
16:02:16 02/03/78
PRINT OF FORMAT: GRAYBEARD
X X X X X X X X X X X X X X X X X X X X



AAAAAAAAAAAAAAAAAAAAAA

XXXXXXXXXXXZZZZZZ

AAAAAAAAAAA   99 99 99

XXXXXXXXXXXX XXXXXX



XXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXX
```

Figure D-4. BLUEBEARD and GRAYBEARD (continued)

```
X                                                                                X


********************************************************************************
********************************************************************************
16:02:16 02/03/78
PRINT OF FORMAT: GRAYBEARD
     PHYS./LOG.                                          AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE    ENTRY  FIELD  ENTRY  SEC
        1   1  X(1)          *
        2   2  X(1)               *
        3   3  X(1)               *
        4   4  X(1)               *
        5   5  X(1)               *
        6   6  X(1)               *
        7   7  X(1)               *
        8   8  X(1)               *
        9   9  X(1)               *
       10  10  X(1)               *
       11  11  X(1)               *
       12  12  X(1)               *
       13  13  X(1)               *
       14  14  X(1)               *
       15  15  X(1)               *
       16  16  X(1)               *
       17  17  X(1)               *
       18  18  X(1)               *
       19  19  X(1)               *
       20  20  X(1)          *
       21  21  A(24)         *
       22  22  X(11)         *
       23  23  9(6)          *
       24  24  A(12)         *
       25  25  9(2)          *
       26  26  9(2)          *
       27  27  9(2)          *
       28  28  X(13)         *
       29  29  X(6)          *
       30  30  X(20)         *
       31  31  X(20)         *
       32  32  X(20)         *
       33  33  X(1)          *
       34  34  X(1)          *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 65
ACS SYNTAX REV 01.01      GRAYBEARD.VS    GRAYBEARD.UP  16:3:3    2/3/78



1
2
3
```

Figure D-4. BLUEBEARD and GRAYBEARD (continued)

```
4       NAME GRAYBEARD
5       *
6       *GRAYBEARD IS A DASHER PROGRAM.  IT READS THE INFOS FILE "INVOICES" AND OUTPUTS
7       *THE CONTENTS OF ITS RECORDS AS HARD COPY.  THE RECORDS IN INVOICES ARE WRITTEN
8       *BY THE PROGRAM "DASHDRVR". THE RECORDS THEMSELVES CONTAIN A PRINT FLAG.  IT I
9       *INITIALLY SET TO "0".  WHEN A RECORD HAS BEEN PRINTED, GRAYBEARD SETS THE
10      *PRINT FLAG TO "1" TO PREVENT FURTHER PRINTING.
11      *
12      *INVOICE RECORDS ARE FILED WITH A SINGLE KEY.  THE KEYS ARE A SET OF
13      *SEQUENTIAL NUMBERS FROM 1 TO 999,999.  WHEN SEARCHING FOR RECORDS TO
14      *PRINT, GRAYBEARD STARTS AT RECORD NO. 1, FINDS IT, PRINTS IT
15      *IF THE PRINT FLAG IS ZERO, THEN MOVES TO RECORD NO. 2 AND REPEATS
16      *THE PROCESS.
17      *
18      *GRAYBEARD HAS A ROW OF DUMMY FIELDS THAT ENABLES THE DASHER
19      *TERMINAL TO STAY ON-LINE BUT INACTIVE, WAITING FOR RECORDS TO PRINT.
20      *GRAYBEARD GOES TO THIS ROW OF DUMMY FIELDS WHENEVER IT HAS
21      *EXHAUSTED THE AVAILABLE RECORDS. IT WAITS THERE UNTIL ANOTHER
22      *RECORD HAS BEEN ADDED.IT THEN RESUMES PRINTING.
23      *
24      *GRAYBEARD'S INERT MODE TAKES ADVANTAGE OF THE INACTIVITY
25      *FEATURE OF THE IDEA MONITOR.  THE PROGRAM SITS INERT AT AN EDIT
26      *FIELD FOR AN INTERVAL MEASURED BY THE INACTIVITY CONSTANT.  AT
27      *THE END OF THE INTERVAL, THE MONITOR SENDS THE PROGRAM TO THE TAG
28      *SPECIFIED BY THE ON NO-ACTIVITY CLAUSE.  AT THE TAG, THE PROGRAM
29      *READS THE FILE AND DECIDES WHETHER TO RESUME PRINTING  OR
30      *CONTINUE WAITING.
31      *
32      *
33      *                        **************************
34      *                        *                        *
35      *                        *        TO USE:         *
36      *                        *  ENTER THIS PROGRAM    *
37      *                        *  FROM THE COMPANION    *
38      *                        *  PROGRAM "BLUEBEARD"   *
39      *                        *                        *
40      *                        **************************
41      *
42      FILE INVOICES
43      KEY FOR INVOICES IS 6 ASCII
44      RECORD FOR INVOICES IS INVREC
45              LENGTH IS 79
46              INCLUDES INVNO 1 6 ASCII
47              INCLUDES IMONTH 7 2 ASCII
48              INCLUDES IDAY 9 2 ASCII
49              INCLUDES IYEAR 11 2 ASCII
50              INCLUDES PO 13 6 ASCII
51              INCLUDES NAME 19 20 ASCII
52              INCLUDES ADDRESS 39 20 ASCII
53              INCLUDES CITY 59 20 ASCII
54              INCLUDES PRTFLG 79 1 ASCII
55              STOP
56      RECORD FOR PASSING IS POINTEREC
57              LENGTH IS 15
58              INCLUDES POINTER 1 6 ASCII
59              INCLUDES SIGNATURE  7  9  ASCII
60              STOP
61
62
63              PROCESS FILLER AT D1 AND NONE
64              PROCESS FILLER AT NONE AND ROUTINE
65              PROCESS FILLER AT NONE AND ROUTINE
66              PROCESS FILLER AT NONE AND ROUTINE
67              PROCESS FILLER AT NONE AND ROUTINE
68              PROCESS FILLER AT NONE AND ROUTINE
69              PROCESS FILLER AT NONE AND ROUTINE
```

Figure D-4. BLUEBEARD and GRAYBEARD (continued)

```
70                PROCESS FILLER AT NONE AND ROUTINE
71                PROCESS FILLER AT NONE AND ROUTINE
72                PROCESS FILLER AT NONE AND ROUTINE
73                PROCESS FILLER AT NONE AND ROUTINE
74                PROCESS FILLER AT NONE AND ROUTINE
75                PROCESS FILLER AT NONE AND ROUTINE
76                PROCESS FILLER AT NONE AND ROUTINE
77                PROCESS FILLER AT NONE AND ROUTINE
78                PROCESS FILLER AT NONE AND ROUTINE
79                PROCESS FILLER AT NONE AND ROUTINE
80                PROCESS FILLER AT NONE AND ROUTINE
81                PROCESS FILLER AT NONE AND ROUTINE
82                PROCESS FILLER AT E20 AND NONE
83       A3#      PROCESS FILLER AT D21 AND NONE
84                PROCESS FILLER AT D22 AND NONE
85                PROCESS INVNO AT D23 AND NONE
86                PROCESS FILLER AT D24 AND NONE
87                PROCESS IMONTH AT D25 AND NONE
88                PROCESS IDAY AT D26 AND NONE
89                PROCESS IYEAR AT D27 AND NONE
90                PROCESS FILLER AT D28 AND NONE
91                PROCESS PO AT D29 AND NONE
92                PROCESS NAME AT D30 AND NONE
93                PROCESS ADDRESS AT D31 AND NONE
94                PROCESS CITY AT D32 AND NONE
95       A2#      PROCESS FILLER AT D33 AND NONE
96       A1#      PROCESS FILLER AT D34 AND NONE
97
98
99                REGISTER POINTER 9(6)
100               REGISTER SELF A(9) GRAYBEARD
101               REGISTER PRTFLG 9
102               REGISTER SIGNATURE A(9)
103               PRIORITY IS 3
104               ON ESCAPE LOGOFF
105               INACTIVITY CONSTANT IS 1
106               ON NO-ACTIVITY ROUTINE
107
108
109
110      *FIELD ONE IS A DISPLAY FIELD.  THIS ALLOWS ALL
111      *AVAILABLE RECORDS TO BE PROMPTLY PRINTED OUT BEFORE
112      *THE INACTIVITY FEATURE OF IDEA IS INVOKED.
113
114      D1:
115               ACCEPT POINTEREC
116               COMPARE SIGNATURE "SIGNATURE"
117               IF EQUAL ROUTINE
118
119      ABORT:
120               MESSAGE PLS ENTER VIA BLUEBEARD. POSITION PRINTHEAD ↑
121      AT ROW 1 COLUMN 1 AND LOG ON AGAIN.
122               QUIT
123
124
125      *FIELDS 2-19 ARE EDIT FIELDS USED FOR WAITING VIA THE
126      *INACTIVITY FEATURE.  IF THERE IS NOTHING TO PRINT BY
127      *THE TIME FIELD 20 IS REACHED, THE PROGRAM OUTPUTS A
128      *SIMULATED FORM FEED (AT A2) AND STARTS OVER.
129
130      E20:
131               RETURN A2
132
133      *THE NEXT FIELD MARKS THE START OF PRINTING.  THE RECORD
134      *HAS ALREADY BEEN FOUND AT "ROUTINE".  HERE IT IS REFILED WITH
135      *THE PRINT FLAG SET, AND A LINE OF LITERAL HEADING INFORMATION
```

Figure D-4. BLUEBEARD and GRAYBEARD (continued)

```
*IS PRINTED.

D21:
        MOVE "1" PRTFLG
        REFILE INVREC USING POINTER
        ADD "1" POINTER POINTER
        PASS POINTEREC
        DISPLAY "DATA GENERAL CORPORATION"
        RETURN

D22:
        DISPLAY "INVOICE NO."
        RETURN

D23:
        DISPLAY INVNO
        RETURN

D24:
        DISPLAY "INVOICE DATE"
        RETURN

D25:
        DISPLAY IMONTH
        RETURN

D26:
        DISPLAY IDAY
        RETURN

D27:
        DISPLAY IYEAR
        RETURN

D28:
        DISPLAY "CUSTOMER P.O."
        RETURN

D29:
        DISPLAY PO
        RETURN

D30:
        DISPLAY NAME
        RETURN

D31:
        DISPLAY ADDRESS
        RETURN

D32:
        DISPLAY CITY
        RETURN


D33:
        LINK USING SELF RETAIN INVOICES

D34:
        MESSAGE GRAYBEARD LOGGED OFF BY DASHER OPERATOR
        QUIT


LOGOFF:
        RETURN A1
```

Figure D-4. BLUEBEARD and GRAYBEARD (continued)

```
202      * *******
203      * ROUTINE
204      * *******
205      *ROUTINE IS EXECUTED ON ENTRY TO THE PROGRAM AND EACH TIME
206      *THE INACTIVITY CONSTANT IS USED UP.  THE PROGRAM HAS BEEN
207      *WAITING AT THE FIRST UNUSED DATABASE RECORD KEY.  HERE
208      *IT CHECKS TO SEE IF THE KEY HAS BEEN USED.  IF IT HAS,
209      *THE PROGRAM READS THE RECORD TO SEE IF IT HAS ALREADY BEEN PRINTED.
210      *IF NOT, CONTROL PASSES TO THE PRINTING ROUTINE AT A3.
211      *IF THE RECORD HAS ALREADY BEEN PRINTED, THE PROGRAM CHECKS
212      *THE NEXT HIGHER KEY.
213      *
214      *IF THE PROGRAM FINDS A KEY UNUSED, IT RETURNS TO ITS WAITING
215      *MODE.
216      *
217      *ROUTINE ALSO CONTAINS A LOOP THAT ENABLES THE PROGRAM TO
218      *SEARCH THROUGH ANY NUMBER OF PRINTED RECORDS WITHOUT USING
219      *ANY PRINTER PAPER. THUS THE OPERATOR CAN INITIATE THE
220      *PROGRAM'S POINTER AT "1" AND LET THE PROGRAM FIND ITS
221      *OWN PLACE IN THE FILE.
222      *
223      *DURING THE ABOVE-MENTIONED LOOP THE PROGRAM PASSES THE POINTER
224      *RECORD ONCE FOR EACH UNSUCCESSFUL SEARCH. WHILE PERHAPS
225      *A BIT FREE AND EASY WITH SYSTEM RESOURCES, THIS SERVES A PURPOSE,
226      *WHICH IS TO ALLOW MONITORING OF THE POINTER (WITH DASHDIAG)
227      *AFTER THE SEARCH HAS ENDED WITH THE POINTER AT AN UNUSED KEY,
228      *AND WITH THE PROGRAM WAITING OUT ITS INACTIVITY CONSTANT.
229
230
231      ROUTINE:
232               FIND INVREC USING POINTER
233               ON-IOERR RET
234               COMPARE PRTFLG "1"
235               IF NOT-EQUAL PRINTIT
236               ADD "1" POINTER POINTER
237               GO TO ROUTINE
238
239      RET:
240               PASS POINTEREC
241               RETURN
242
243      PRINTIT:
244               RETURN A3
245
246               FINISH
 FORMAT NOT LINKED
```

Figure D-4. BLUEBEARD and GRAYBEARD (continued)

GRAYBEARD

FIELD NAME	PHYS./LOG. FIELD#		DESCRIPTION	DISP	EDIT	OUTPUT	AUTO-DUPE	REQ. ENTRY	FULL FIELD	AUTO-ENTRY SEC
FILLER	1	1	X(1)	*						
FILLER	2	2	X(1)		*					
FILLER	3	3	X(1)		*					
FILLER	4	4	X(1)		*					
FILLER	5	5	X(1)		*					
FILLER	6	6	X(1)		*					
FILLER	7	7	X(1)		*					
FILLER	8	8	X(1)		*					
FILLER	9	9	X(1)		*					
FILLER	10	10	X(1)		*					
FILLER	11	11	X(1)		*					
FILLER	12	12	X(1)		*					
FILLER	13	13	X(1)		*					
FILLER	14	14	X(1)		*					
FILLER	15	15	X(1)		*					
FILLER	16	16	X(1)		*					
FILLER	17	17	X(1)		*					
FILLER	18	18	X(1)		*					
FILLER	19	19	X(1)		*					
FILLER	20	20	X(1)	*						
FILLER	21	21	X(24)	*						
FILLER	22	22	X(11)	*						
INVNO	23	23	9(6)	*						
FILLER	24	24	X(12)	*						
IMONTH	25	25	9(2)	*						
IDAY	26	26	9(2)	*						
IYEAR	27	27	9(2)	*						
FILLER	28	28	X(13)	*						
PO	29	29	X(6)	*						
NAME	30	30	X(20)	*						
ADDRESS	31	31	X(20)	*						
CITY	32	32	X(20)	*						
FILLER	33	33	X(1)	*						
FILLER	34	34	X(1)	*						

Figure D-4. BLUEBEARD and GRAYBEARD (continued)

```
13:19:54 02/03/78
PRINT OF FORMAT: DASHDIAG

     ******************          ************          **********
     *     999999     *          * 99 99 99 *          * XXXXXX *
     * INVOICE NUMBER *          *   DATE   *          *   PO   *
     ******************          ************          **********


************************
* XXXXXXXXXXXXXXXXXXXX *
*        NAME          *
************************
                          ************************
                          * XXXXXXXXXXXXXXXXXXXX *
                          *       ADDRESS        *
                          ************************
                                                  ************************
                                                  * XXXXXXXXXXXXXXXXXXXX *
                                                  *   CITY  STATE  ZIP   *
                                                  ************************
            PRINT FLAG  *****       ************
       (1 => RECORD HAS  * 9 *      * 999999   *          ************
  ALREADY BEEN PRINTED)  *****       * POINTER *          * AGAIN  X *
                                     ************          ************



********************************************************************************
********************************************************************************
13:19:54 02/03/78
PRINT OF FORMAT: DASHDIAG
     PHYS./LOG.                                          AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
        1   1  9(6)                *
        2   2  9(2)           *
        3   3  9(2)           *
        4   4  9(2)           *
        5   5  X(6)           *
        6   6  X(20)          *
        7   7  X(20)          *
        8   8  X(20)          *
        9   9  9(1)           *      *
        10  10 9(6)           *      *
        11  11 X(1)                 *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 23
AOS SYNTAX REV 01.01      DASHDIAG.VS    DASHDIAG.UP  13:20:40   2/3/78



1
2
3
4       NAME DASHDIAG
5       *
6       *DASHDIAG IS USED FOR "DIAGNOSING" THE GROUP OF PROGRAMS
7       *USED FOR PRINTING ON THE DGC DASHER TERMINAL. IT READS
8       *THE "INVOICES" DATA BASE RECORD WRITTEN BY "DASHDRVR" AND
9       *DISPLAYS THE DATA FOUND THERE. THIS INCLUDES THE PRINT
10      *FLAG.  THE PROGRAM GIVES THE OPERATOR THE OPTION OF
11      *CHANGING THE PRINT FLAG. THUS IF THE FLAG HAS THE VALUE
12      *OF "1" AND THE OPERATOR WANTS TO REPRINT THE RECORD,
13      *THE VALUE MAY BE SET TO "0".
14      *
```

Figure D-5. DASHDIAG

```
15      *THE PROGRAM ALSO READS THE COMMON FILE FOR THE RECORD FOR
16      *PASSING USED BY THE DASHER TERMINAL USING "DASHPRNT". FOR THE
17      *PURPOSES OF THE PRESENT PROGRAM, THE DASHER'S LINE NUMBER IS 08.
18      *THUS, THE DASHER PASSING RECORD IS FILED UNDER THE KEYS
19      *??PASSING??, 08. THE DASHER USES ITS RECORD FOR PASSING
20      *TO KEEP ITS PLACE AMONG THE DATA BASE RECORDS IT IS
21      *PRINTING.  THIS RECORD FOR PASSING MAY BE RESET TO
22      *"1" TO ALLOW THE PROGRAM TO REPRINT RECORDS ALREADY PRINTED.
23      *
24      FILES INVOICES, COMMON
25      KEY FOR INVOICES IS 6 ASCII
26      RECORD FOR INVOICES IS INVREC
27           LENGTH IS 79
28           INCLUDES INVNO 1 6 ASCII
29           INCLUDES IMONTH 7 2 ASCII
30           INCLUDES IDAY 9 2 ASCII
31           INCLUDES IYEAR 11 2 ASCII
32           INCLUDES PO 13 6 ASCII
33           INCLUDES NAME 19 20 ASCII
34           INCLUDES ADDRESS 39 20 ASCII
35           INCLUDES CITY 59 20 ASCII
36           INCLUDES PRTFLG 79 1 ASCII
37           STOP
38
39      KEY FOR COMMON IS 13 ASCII
40      SUBINDEX FOR COMMON IS LEVEL1
41      KEY FOR LEVEL1 IS 2 ASCII
42      RECORD FOR LEVEL1 IS PASSREC
43      LENGTH IS 15
44           INCLUDES POINTER  1  6  ASCII
45           STOP
46
47      PROCESS INVNO AT NONE AND E1
48      PROCESS IMONTH AT D2 AND NONE
49      PROCESS IDAY AT D3 AND NONE
50      PROCESS IYEAR AT D4 AND NONE
51      PROCESS PO AT D5 AND NONE
52      PROCESS NAME AT D6 AND NONE
53      PROCESS ADDRESS AT D7 AND NONE
54      PROCESS CITY AT D8 AND NONE
55      PROCESS PRTFLG AT D9 AND E9
56      PROCESS POINTER AT D10 AND E10
57      PROCESS FILLER AT NONE AND E11
58
59
60      REGISTER FIELD 9(2)
61      REGISTER SELF A(8) DASHDIAG
62
63
64      E1:
65                   STORE INVNO
66                   FIND INVREC USING INVNO
67                   ON-IOERR D1B
68                   RETURN
69
70
71      D1B:
72                   MESSAGE NO RECORD OF INVOICE, TRY AGAIN.
73                   RETURN USING FIELD
74
75      D2:
76                   DISPLAY IMONTH
77                   RETURN
78
```

Figure D-5. DASHDIAG (continued)

```
79      D3:
80                      DISPLAY IDAY
81                      RETURN
82
83      D4:
84                      DISPLAY IYEAR
85                      RETURN
86
87      D5:
88                      DISPLAY PO
89                      RETURN
90
91      D6:
92                      DISPLAY NAME
93                      RETURN
94
95      D7:
96                      DISPLAY ADDRESS
97                      RETURN
98
99      D8:
100                     DISPLAY CITY
101                     RETURN
102
103     D9:
104                     DISPLAY PRTFLG
105                     RETURN USING FIELD
106
107     E9:
108                     STORE PRTFLG
109                     REFILE INVREC USING INVNO
110                     RETURN
111
112     D10:
113                     FIND THE PASSREC USING "??PASSING??", "08"
114                     DISPLAY POINTER
115                     RETURN USING FIELD
116
117     E10:
118                     STORE POINTER
119                     REFILE PASSREC USING "??PASSING??", "08"
120                     RETURN
121
122     E11:
123                     LINK USING SELF AND RETAIN INVOICES, COMMON
124
125     FINISH
 FORMAT NOT LINKED

DASHDIAG

FIELD      PHYS./LOG.                                    AUTO-  REQ.   FULL   AUTO-
NAME         FIELD#   DESCRIPTION DISP EDIT OUTPUT DUPE   ENTRY FIELD ENTRY SEC

INVNO          1    1  9(6)               *
IMONTH         2    2  9(2)          *
IDAY           3    3  9(2)          *
IYEAR          4    4  9(2)          *
PO             5    5  X(6)          *
NAME           6    6  X(20)         *
ADDRESS        7    7  X(20)         *
CITY           8    8  X(20)         *
PRTFLG         9    9  9(1)          *     *
POINTER        10   10 9(6)          *     *
FILLER         11   11 X(1)               *
```

Figure D-5. DASHDIAG (continued)

```
13:25:01 02/03/78
PRINT OF FORMAT: HSPA7

  PATIENT NUMBER            PATIENT NAME              PATIENT TYPE

       9999             XXXXXXXXXXXXXXXXXX               A

 NUMBER OF CHARGES ZZZ9        AMOUNT OF CHARGES    +$$$$$9.99
-------------------------------------------------------------------------------
   DEPT #    CHARGE #     COST        DESCRIPTION          DATE        TIME
@
     99         99     +$$$9.99  XXXXXXXXXXXXXXXXXXX     99/99/99     99:99


@
-------------------------------------------------------------------------------
   CHARGES ADDED    AMOUNT         TOTAL CHARGES      TOTAL AMOUNT
      ZZZ9        +$$$$$9.99          ZZZ9            +$$$$$9.99

               IF YOU HAVE MORE CHARGES STRIKE THE RETURN KEY,
                 OTHERWISE STRIKE ANY KEY AND THE RETURN KEY.
                                X


*******************************************************************************
*******************************************************************************
*******************************************************************************
*******************************************************************************
13:25:01 02/03/78
PRINT OF FORMAT: HSPA7
     PHYS./LOG.                                        AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
        1   1  9(4)               *
        2   2  X(20)         *
        3   3  A(1)           *
        4   4  9(4)           *
        5   5  S9(5).9(2)     *
        6   6  9(2)                *                                    *
        7   7  9(2)                *                                    *
        8   8  S9(3).9(2)     *    *
        9   9  X(20)          *
       10  10  9(2)           *
       11  11  9(2)           *
       12  12  9(2)           *
       13  13  9(2)           *
       14  14  9(2)           *
       15  15  9(4)           *
       16  16  S9(5).9(2)     *
       17  17  9(4)           *
       18  18  S9(5).9(2)     *
       19  19  X(1)                *
LINKED TO FORMAT: HMENU
FIRST LINE USED: 1
LAST LINE USED: 23
AOS SYNTAX REV 01.01      HSPA7.VS    HSPA7.UF  13:25:36   2/3/78
```

Figure D-6. HSPA7

```
1                       NAME HSPA7.LP
2                 FILE HSPDB,HSPPM,HSPCH
3                 SUBINDEX FOR HSPDB IS AFILE
4                 KEY FOR AFILE IS 4 ASCII
5                 KEY FOR HSPPM IS 4 ASCII
6                 KEY FOR HSPDB IS 1 ASCII
7                 KEY FOR HSPCH IS 4 ASCII
8                 DUPLICATES ARE COUNTED IN DUP1
9                 RECORD FOR HSPPM IS PATREC
10                BUFFER LENGTH IS 106
11                      INCLUDES PATNO  1  4  ASCII
12                      INCLUDES PATNAM 19  20  ASCII
13                      INCLUDES PATTYP  76  1  ASCII
14                      INCLUDES PTCHRG  82  4  ASCII
15                      INCLUDES PTCSUM  86  8  ASCII
16                STOP
17                RECORD FOR AFILE IS DPTREC
18                LENGTH IS 31
19                      INCLUDES DEPT  1  2  ASCII
20                      INCLUDES CHRG# 3  2  ASCII
21                      INCLUDES COST  5  6  ASCII
22                      INCLUDES DESCRP  11  20  ASCII
23                STOP
24                RECORD FOR HSPCH IS CHARGE
25                LENGTH IS 45
26                      INCLUDES PATNO 1  4  ASCII
27                      INCLUDES DEPT  5  2  ASCII
28                      INCLUDES CHRG# 7 2 ASCII
29                      INCLUDES COST 9  6  ASCII
30                      INCLUDES DESCRP 15  20  ASCII
31                      INCLUDES MONTH  35  2  ASCII
32                      INCLUDES DAY    37  2  ASCII
33                      INCLUDES YEAR   39  2  ASCII
34                      INCLUDES HOURS  41  2  ASCII
35                      INCLUDES MINS   43  2  ASCII
36                      INCLUDES PATTYP 45  1  ASCII
37                STOP
38                ON SCREEN PRINT
39                REGISTER DUMFLD   X(4) ABCD
40                REGISTER FIELD 99
41                REGISTER ONE 9 1
42                REGISTER DUP1 9999
43                REGISTER HOLD 9999
44                REDESIGNATE HOLD
45                   DEPT 1 2
46                   CHRG# 3 2
47                STOP
48                REGISTER ZERO 9
49                REGISTER BLANK X
50                REGISTER C X C
51                REGISTER HMENU XXXXX HMENU
52                ****************************************
53                PROCESS PATNO AT NONE AND A1
54                PROCESS PATNAM AT A2 AND NONE
55                PROCESS PATTYP AT A3 AND NONE
56                PROCESS PTCHRG AT A4 AND NONE
57                PROCESS PTCSUM AT A5 AND NONE
58                PROCESS FILLER AT NONE AND A8
59                PROCESS FILLER AT NONE AND A9
60                PROCESS COST AT A10 AND A10A
61                PROCESS DESCRP AT A11 AND NONE
62                PROCESS MONTH AT A11A AND NONE
63                PROCESS DAY AT A11B AND NONE
64                PROCESS YEAR AT A11C AND NONE
65                PROCESS HOURS AT A11D AND NONE
```

Figure D-6. HSPA7 (continued)

```
66              PROCESS MINS AT A11E AND NONE
67              PROCESS NUM1 AT A12 AND NONE
68              PROCESS AMT AT A13 AND NONE
69              PROCESS CHARG AT A14 AND NONE
70              PROCESS NUM3 AT A15 AND NONE
71              PROCESS ANS1 AT NONE AND A16
72              ****************************************
73              A1:     STORE PATNO
74                      FIND AND HOLD PATREC USING PATNO
75                      ON-IOERR MSG1
76                      RETURN
77              A2:     DISPLAY PATNAM
78                      RETURN
79              A3:     DISPLAY PATTYP
80                      RETURN
81              A4:     DISPLAY PTCHRG
82                      RETURN
83              A5:     DISPLAY PTCSUM
84                      MOVE ZERO AMT
85                      MOVE ZERO CHARG
86                      RETURN
87              A8:     STORE DEPT
88                      COMPARE DEPT "00"
89                      IF EQUAL RTN15
90                      RETURN
91              A9:     STORE CHRG#
92                      FIND DPTREC USING C,HOLD
93                      ON-IOERR MSG2
94                      RETURN
95              A10:    DISPLAY COST
96                      RETURN
97              A10A:   STORE COST
98                      ADD COST AMT AMT
99                      ADD ONE CHARG CHARG
100                     RETURN 10
101             A11:    DISPLAY DESCRP
102                     RETURN 8
103             A11A:   DISPLAY MONTH
104                     RETURN
105             A11B:   DISPLAY DAY
106                     RETURN
107             A11C:   DISPLAY YEAR
108                     RETURN
109             A11D:   DISPLAY HOURS
110                     RETURN
111             A11E:   DISPLAY MINS
112                     FILE-NEW CHARGE USING PATNO
113                     ON-IOERR MSG3
114                     RETURN
115             A12:    DISPLAY CHARG
116                     ADD CHARG PTCHRG PTCHRG
117                     RETURN
118             A13:    DISPLAY AMT
119                     ADD AMT PTCSUM PTCSUM
120                     RETURN
121             A14:    DISPLAY PTCHRG
122                     RETURN
123             A15:    DISPLAY PTCSUM
124                     REFILE PATREC USING PATNO
125                     ON-IOERR MSG4
126                     RETURN
127             A16:    STORE ANS1
128                     COMPARE ANS1 BLANK
129                     IF EQUAL NEXT
130                     LINK USING HMENU RETAIN HSPDB,HSPPM,HSPCH
```

Figure D-6. HSPA7 (continued)

```
131             MSG1:   COMPARE IOERR "94"
132                     IF EQUAL RLOC
133                     MESSAGE <7> PATIENT NOT ON FILE
134             RTN1:   RETURN 1
135             RLOC:   MESSAGE <7> RECORD IN USE-- TRY LATER.
136                     GO TO RTN1
137             MSG2:   MESSAGE <7> DEPARTMENT-CHARGE NUMBER NOT ON FILE
138                     RETURN 6
139             MSG3:   MESSAGE <7> CHARGE FILE ERROR CALL SUPERVISOR
140                     QUIT
141             MSG4:   MESSAGE <7> PATIENT MASTER REFILE ERROR - CALL SUPERVISOR
142                     QUIT
143             RTN15:  RETURN 15
144             NEXT:   RESTART
145             PRINT:  MESSAGE <10><21>
146             RETURN USING FIELD
147             FINISH
 FORMAT LINKED TO HMENU

HSPA7

FIELD     PHYS./LOG.                                    AUTO- REQ.  FULL  AUTO-
NAME        FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

PATNO         1   1  9(4)              *
PATNAM        2   2  X(20)        *
PATTYP        3   3  X(1)          *
PTCHRG        4   4  9(4)          *
PTCSUM        5   5  S9(5).9(2)    *
FILLER        6   6  9(2)              *                                  *
FILLER        7   7  9(2)              *                                  *
COST          8   8  S9(3).9(2)    *   *
DESCRP        9   9  X(20)          *
MONTH        10  10  9(2)          *
DAY          11  11  9(2)          *
YEAR         12  12  9(2)          *
HOURS        13  13  9(2)          *
MINS         14  14  9(2)          *
NUM1         15  15  9(4)          *
AMT          16  16  S9(5).9(2)    *
CHARG        17  17  9(4)          *
NUM3         18  18  S9(5).9(2)    *
ANS1         19  19  X(1)              *
```

Figure D-6. HSPA7 (continued)

```
16:20:33 02/03/78
PRINT OF FORMAT: BIGFOOT


                    LARGE-LETTER DISPLAY

MESSAGE XXXXXXXX

á    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

á

          ENTER "Y" FOR LINE PRINTER OUTPUT


*******************************************************************************
                         BIGFOOT
                         *******

BIGFOOT ACCEPTS AN OPERATOR MESSAGE AND SCROLLS IT AS A LARGE-LETTER
DISPLAY. IT WILL PRODUCE LINE PRINTER OUTPUT OF THE LARGE-LETTER
MESSAGE AT THE OPTION OF THE OPERATOR.

                         KEY TO FIELDS
                         *************

1. ACCEPTS INPUT FROM THE KEYBOARD.
   CHECKS INPUT AGAINST ALPHABET AVAILABLE IN BIGFOOT REPERTOIRE.
   WHEN INPUT IS ACCEPTABLE, INITIATES PRINTING AND INITIALIZES
   SCROLL VARIABLES.

2. SCROLLS MESSAGE IN LETTERS 7 LINES HIGH

3. TIDIES UP PRINTING RECORDS IN A DISPLAY ROUTINE.
   RETURNS TO SEE WHETHER TO ORDER LINE PRINTER OUTPUT
   QUEUES CLI COMMAND TO QPRINT IF SO ORDERED.
*******************************************************************************
16:20:33 02/03/78
PRINT OF FORMAT: BIGFOOT
     PHYS./LOG.                                    AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT     DUPE   ENTRY  FIELD  ENTRY  SEC
         1   1  X(8)               *
         2   2  X(70)         *
         3   3  A(1)           *     *
LINKED TO FORMAT: BIGFOOT
FIRST LINE USED: 1
LAST LINE USED: 20
AOS SYNTAX REV 01.01     BIGFOOT.VS   BIGFOOT.UP  16:22:20   2/3/78
```

Figure D-7. BIGFOOT

```
1        **************
2        NAME BIGFOOT *
3        **************
4
5        RECORD FOR PRINTING IS PRINTREC
6        LENGTH IS 70
7             INCLUDES MESSAGE  1  70 ASCII
8             STOP
9        RECORD FOR PRINTING IS DATEREC
10       LENGTH IS 8
11            INCLUDES DATE  1  8  ASCII
12            STOP
13       RECORD FOR PRINTING IS ENDSCROLL
14       LENGTH IS 1
15            INCLUDES "@"  1  1 ASCII
16            STOP
17
18       ********************
19       * REGISTER SECTION *
20       ********************
21
22       REGISTER SCRLLKNT   9(1)
23       REGISTER FLDPTR     9(1)
24       REGISTER LETTER     X(1)
25       REGISTER FIELD      9(2)
26       REGISTER PTR        9(3)
27       REGISTER DATE       X(8)  00/00/00
28       REDESIGNATE DATE
29            MONTH  1  2
30            DAY    4  2
31            YEAR   7  2
32            STOP
33       REGISTER PTITLE     X(6) PTITLE
34       REGISTER CLI    X(20) XEQ PRINTF PTITLE 99
35       REDESIGNATE CLI
36               CRTNO  19  2
37               STOP
38       REGISTER CRT    9(2)
39       COPY ALPHASOUP
40       *RDOS FILE ALPHASOUP
41       REGISTER MESSAGE X(70)
42       *USED TO SCROLL A DISPLAY OF LARGE LETTERS
43       REDESIGNATE MESSAGE
44            FLD1   1  7
45            FLD2  10  7
46            FLD3  19  7
47            FLD4  28  7
48            FLD5  37  7
49            FLD6  46  7
50            FLD7  55  7
51            FLD8  64  7
52            STOP
53
54       REGISTER JERRYHALL X(8)
55       REDESIGNATE JERRYHALL
56            L1  1 1
57            L2  2 1
58            L3  3 1
59            L4  4 1
60            L5  5 1
61            L6  6 1
62            L7  7 1
63            L8  8 1
64            STOP
```

Figure D-7. BIGFOOT (continued)

```
65
66        TABLE LETTERS
67        *USED TO COLLECT INPUT FOR LARGE-CHARACTER
68        *MESSAGE; AS, 'STORE L1'
69
70        L1
71        L2
72        L3
73        L4
74        L5
75        L6
76        L7
77        L8
78        ENDTABLE
79
80        TABLE DISPFLD
81        *USED TO ASSEMBLE VALUES FOR A SCROLL LINE
82
83        FLD1
84        FLD2
85        FLD3
86        FLD4
87        FLD5
88        FLD6
89        FLD7
90        FLD8
91        ENDTABLE
92
93
94        TABLE ALPHABET
95        *USED TO LOOK UP INPUT CHARACTERS FOR MESSAGE
96
97
98        " "     *ALPH1
99        "0"     *ALPH2
100       "1"     *ALPH3
101       "2"     *ALPH4
102       "3"     *ALPH5
103       "4"     *ALPH6
104       "5"     *ALPH7
105       "6"     *ALPH8
106       "7"     *ALPH9
107       "8"     *ALPH10
108       "9"     *ALPH11
109       "A"     *ALPH12
110       "B"     *ALPH13
111       "C"     *ALPH14
112       "D"     *ALPH15
113       "E"     *ALPH16
114       "F"     *ALPH17
115       "G"     *ALPH18
116       "H"     *ALPH19
117       "I"     *ALPH20
118       "J"     *ALPH21
119       "K"     *ALPH22
120       "L"     *ALPH23
121       "M"     *ALPH24
122       "N"     *ALPH25
123       "O"     *ALPH26
124       "P"     *ALPH27
125       "Q"     *ALPH28
126       "R"     *ALPH29
127       "S"     *ALPH30
128       "T"     *ALPH31
129       "U"     *ALPH32
130       "V"     *ALPH33
```

Figure D-7. BIGFOOT (continued)

```
131     "W"     *ALPH34
132     "X"     *ALPH35
133     "Y"     *ALPH36
134     "Z"     *ALPH37
135     "."     *ALPH38
136     ENDTABLE
137     TABLE ALPHALPHA
138     "       "
139     "       "
140     "       "
141     "       "
142     "       "
143     "       "
144     "       "
145     "  000  "
146     " 0   0 "
147     "0    0 0"
148     "0  0  0"
149     "0 0   0"
150     " 0   0 "
151     "  000  "
152     " 11111 "
153     "1   11 "
154     "    11 "
155     "    11 "
156     "    11 "
157     "    11 "
158     "1111111"
159     " 22222 "
160     "22   22"
161     "     22"
162     "    22 "
163     "  22   "
164     " 22    "
165     "2222222"
166     " 333333"
167     "     33"
168     "    33 "
169     "  33   "
170     "    33 "
171     "33   33"
172     " 33333 "
173     "44   44"
174     "44   44"
175     "44   44"
176     "4444444"
177     "     44"
178     "     44"
179     "     44"
180     "5555555"
181     "55     "
182     "55     "
183     " 55555 "
184     "     55"
185     "5    55"
186     " 55555 "
187     " 66666 "
188     "66   66"
189     "66     "
190     "666666 "
191     "66   66"
192     "66   66"
193     " 66666 "
```

Figure D-7. BIGFOOT (continued)

```
194      "7777777"
195      "     77"
196      "    77 "
197      "   77  "
198      "  77   "
199      " 77    "
200      "77     "
201      " 88888 "
202      "88   88"
203      "88   88"
204      " 88888 "
205      "88   88"
206      "88   88"
207      " 88888 "
208      " 99999 "
209      "99   99"
210      "99   99"
211      " 999999"
212      "     99"
213      "99   99"
214      " 99999 "
215      "  AAA  "
216      " AA AA "
217      "AA   AA"
218      "AAAAAAA"
219      "AA   AA"
220      "AA   AA"
221      "AA   AA"
222      "BBBBB "
223      "BB   BB"
224      "BB   BB"
225      "BBBBB "
226      "BB   BB"
227      "BB   BB"
228      "BBBBB "
229      "   CCC "
230      " CC  CC"
231      "CC     "
232      "CC     "
233      "CC   CC"
234      " CC  CC"
235      "  CCCC "
236      "DDDD  "
237      "DD  DD "
238      "DD   DD"
239      "DD   DD"
240      "DD   DD"
241      "DD  DD "
242      "DDDD  "
243      "EEEEEEE"
244      "EE     "
245      "EE     "
246      "EEEE  "
247      "EE     "
248      "EE     "
249      "EEEEEEE"
250      "FFFFFFF"
251      "FF     "
252      "FF     "
253      "FFFF  "
254      "FF     "
255      "FF     "
256      "FF     "
```

Figure D-7. BIGFOOT (continued)

```
257     " GGGGG "
258     "GG   GG"
259     "GG     "
260     "GG GGGG"
261     "GG   GG"
262     "GG   GG"
263     " GGGGG "
264     "HH   HH"
265     "HH   HH"
266     "HH   HH"
267     "HHHHHHH"
268     "HH   HH"
269     "HH   HH"
270     "HH   HH"
271     "IIIIII "
272     "  II   "
273     "  II   "
274     "  II   "
275     "  II   "
276     "  II   "
277     "IIIIII "
278     " JJJJJJ"
279     "   JJ  "
280     "   JJ  "
281     "   JJ  "
282     "JJ JJ  "
283     " JJJJ  "
284     "  JJ   "
285     "KK   KK"
286     "KK  KK "
287     "KK KK  "
288     "KKKK   "
289     "KK KK  "
290     "KK  KK "
291     "KK   KK"
292     "LL     "
293     "LL     "
294     "LL     "
295     "LL     "
296     "LL     "
297     "LL     "
298     "LLLLLLL"
299     "MM   MM"
300     "MM   MM"
301     "M M M M"
302     "M  M  M"
303     "M  M  M"
304     "M     M"
305     "M     M"
306     "N     N"
307     "NN    N"
308     "N N   N"
309     "N  N  N"
310     "N   N N"
311     "N    NN"
312     "N     N"
313     "  000  "
314     " 00 00 "
315     "00   00"
316     "00   00"
317     "00   00"
318     " 00 00 "
319     "  000  "
```

Figure D-7. BIGFOOT (continued)

```
320      "PPPPPP  "
321      "PP    PP"
322      "PP    PP"
323      "PPPPPP  "
324      "PP      "
325      "PP      "
326      "PP      "
327      "  QQQ   "
328      " Q   Q  "
329      "Q     Q "
330      "Q     Q "
331      "Q   Q Q "
332      " Q  QQ  "
333      "  QQQ Q "
334      "RRRRRR  "
335      "RR    RR"
336      "RR    RR"
337      "RRRRR   "
338      "RR RR   "
339      "RR  RR  "
340      "RR   RR "
341      " SSSSS  "
342      "SS    SS"
343      " SS     "
344      "   SS   "
345      "    SS  "
346      "SS    SS"
347      " SSSSS  "
348      "TTTTTTT "
349      "  TT    "
350      "  TT    "
351      "  TT    "
352      "  TT    "
353      "  TT    "
354      "  TT    "
355      "UU    UU"
356      "UU    UU"
357      "UU    UU"
358      "UU    UU"
359      "UU    UU"
360      "UU    UU"
361      " UUUUU  "
362      "VV    VV"
363      "VV    VV"
364      " VV  VV "
365      " VV  VV "
366      "  VVV   "
367      "   V    "
368      "   V    "
369      "W      W"
370      "W      W"
371      "W  W   W"
372      "W  W   W"
373      "W W W W "
374      "WW    WW"
375      "WW    WW"
376      "XX    XX"
377      " XX  XX "
378      "  XXX   "
379      "   X    "
380      "  XXX   "
381      " XX  XX "
382      "XX    XX"
383      "YY    YY"
```

Figure D-7. BIGFOOT (continued)

```
384      " YY YY "
385      "  Y Y  "
386      "   Y   "
387      "   Y   "
388      "   Y   "
389      "   Y   "
390      "ZZZZZZZ"
391      "     ZZ"
392      "    ZZ "
393      "   ZZ  "
394      " ZZ    "
395      "ZZ     "
396      "ZZZZZZZ"
397      "       "
398      "       "
399      "       "
400      "       "
401      "       "
402      "PPP    "
403      "PPP    "
404      ENDTABLE
405
406
407      ********************
408      * PROCESS SECTION *
409      ********************
410
411              PROCESS FILLER AT NONE AND E1
412              PROCESS FILLER AT D2 AND NONE
413      A3#     PROCESS COMMAND AT D3 AND E3
414
415      ************************
416      * EXECUTABLE SECTION *
417      ************************
418
419      E1:
420              STORE JERRYHALL
421              MOVE "1" FLDPTR
422
423      E1A:
424              MOVE LETTERS (FLDPTR) TO LETTER *CHECK MESSAGE AGAINST
425              LOOKUP IN ALPHABET LETTER       *LEGITIMATE ALPHABET
426              COMPARE ENTRY "00"
427              IF EQUAL NG
428              ADD "1" FLDPTR FLDPTR
429              COMPARE FLDPTR "8"
430              IF GREATER E1B                  *MESSAGE IS NOW IN 'LETTERS'
431              GO TO E1A                       *(SEE 'ALPHASOUP')
432
433      NG:
434              MESSAGE CHARACTER NOT IN CURRENT ALPHABET, PLEASE RE-ENTER.
435              RETURN 1
436
437
438      E1B:
439              MOVE "1" SCRLLKNT               *INITIALIZE SCROLL VARIABLES
440              MOVE "1" FLDPTR
441              INITIATE PRINTING USING PTITLE
442              RETURN                          *GO TO SCROLL FIELD
443
444
```

Figure D-7. BIGFOOT (continued)

```
D2:                                          *THE SCROLL FIELD
            MOVE LETTERS (FLDPTR) TO LETTER
            LOOKUP ALPHABET LETTER
            SUBTRACT "1" ENTRY PTR           *SYNCHRONIZE 'SUBSCRIPTS' FOR
            MULTIPLY PTR "7" PTR             *2-D ARRAY LOOKUP AND
            ADD SCRLLKNT PTR PTR             *RETRIEVE PART OF LETTER

            MOVE ALPHALPHA (PTR) TO DISPFLD (FLDPTR)

            ADD "1" FLDPTR FLDPTR            *LOOP TERMINATION LOGIC
            COMPARE FLDPTR "9"               *STEP POINTER THROUGH 8 FIELDS
            IF LESS D2                       *OF DISPLAY

            DISPLAY MESSAGE                  *SCROLL 7 LINES OF DISPLAY
            PRINT PRINTREC USING PTITLE      *AND WRITE TO PRINT FILE
            ADD "1" SCRLLKNT SCRLLKNT
            COMPARE SCRLLKNT "7"
            IF GREATER D2A
            RETURN

D2A:
            RETURN A3                        *FIELD A3 IS OUTSIDE
                                             *THE SCROLL AREA

D3:
            PRINT ENDSCROLL USING PTITLE     *NE-TEN UP, YOU'RE OUT
            PRINT DATEREC USING PTITLE       *OF THE SCROLL AREA
            TERMINATE PRINTING USING PTITLE
            RETURN USING FIELD


E3:
            STORE COMMAND                    *DOES THIS GUY WANT
            COMPARE COMMAND "Y"              *LINE PRINTER OUTPUT OR NOT?
            IF NOT-EQUAL E3A
            MOVE CRT CRTNO
                     ↑
****WARNING--FIELDS OF INCOMPATIBLE TYPE. ALPHA FUNCTION ASSUMED

            QUEUE CLI
E3A:
            RETURN                           *RESTARTS USING FORMAT LINK

            FINISH
 FORMAT LINKED TO BIGFOOT

BIGFOOT

FIELD      PHYS./LOG.                               AUTO- REG.  FULL  AUTO-
NAME         FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

FILLER        1   1   X(8)             *
FILLER        2   2   X(70)       *
COMMAND       3   3   X(1)        *    *
14:40:32 02/02/78
PRINT OF FORMAT: PTITLE




a    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
a
```

Figure D-7. BIGFOOT (continued)

```
                                DATE  XXXXXXXX



                        IDEA INTERACTIVE DATA ENTRY-ACCESS
14:40:32 02/02/78
PRINT OF FORMAT: PTITLE
     PHYS./LOG.                                          AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE    ENTRY  FIELD  ENTRY  SEC
        1    0  X(70)
        2    0  X(8)
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 38
```

Figure D-7. BIGFOOT (continued)

```
16:25:48 02/03/78
PRINT OF FORMAT: CRAIGS
                          EXAMPLE FORMAT

            DATA GENERAL CUMULATIVE COMPUTER SHIPMENTS
                         DATA COLLECTION


           GRAPHIC CHARACTERS : XXXX


             TITLE CENTERED BETWEEN CARETS SHOWN BELOW

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          ↑                                               ↑

         ENTER 6 YEARS ON FIRST LINE BELOW THE DOTS, THEN DATA FOR EACH YEAR
             ....        ....        ....        ....        ....        ....
             ZZZ9        ZZZ9        ZZZ9        ZZZ9        ZZZ9        ZZZ9

           ZZZ,ZZ9     ZZZ,ZZ9     ZZZ,ZZ9     ZZZ,ZZ9     ZZZ,ZZ9     ZZZ,ZZ9


HIGHEST NUMBER TO APPEAR ON THE GRAPH : ZZZ,ZZ9


********************************************************************************
********************************************************************************
16:25:48 02/03/78
PRINT OF FORMAT: CRAIGS
     PHYS./LOG.                                    AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT     DUPE   ENTRY  FIELD  ENTRY  SEC
        1   1  X(4)          *     *                                      *
        2   2  X(47)               *
        3   3  9(4)               *                        *
        4   4  9(4)               *                        *
        5   5  9(4)               *                        *
        6   6  9(4)               *                        *
        7   7  9(4)               *                        *
        8   8  9(4)               *                        *
        9   9  9(6)          *    *
       10  10  9(6)          *    *
       11  11  9(6)          *    *
       12  12  9(6)          *    *
       13  13  9(6)          *    *
       14  14  9(6)          *    *
       15  15  9(6)          *    *
LINKED TO FORMAT: BARGRAPH
FIRST LINE USED: 1
LAST LINE USED: 23
AOS SYNTAX REV 01.01      CRAIGS.VS   CRAIGS.UP  16:26:20   2/3/78
```

Figure D-8. CRAIGS and BARGRAPH

```
1
2
3       *
4       *
5       *       THIS PROGRAM COLLECTS DATA FOR THE BARGRAPH PROGRAM AND THEN
6       *       LINKS TO IT
7       *
8       *
9
10              NAME GRAPHDATA
11
12              RECD FOR PASSING IS PASS-REC
13                      LENGTH IS 129
14                      INCL YEAR-1972 1 6 ASCII
15                      INCL YEAR-1973 8 6 ASCII
16                      INCL YEAR-1974 15 6 ASCII
17                      INCL YEAR-1975 22 6 ASCII
18                      INCL YEAR-1976 29 6 ASCII
19                      INCL YEAR-1977 36 6 ASCII
20                      INCL TITLE 43 47 ASCII
21                      INCL GRAPHIC 90 4 ASCII
22                      INCL MAX 94 6 ASCII
23                      INCL YEAR1 100 4 ASCII
24                      INCL YEAR2 105 4 ASCII
25                      INCL YEAR3 110 4 ASCII
26                      INCL YEAR4 115 4 ASCII
27                      INCL YEAR5 120 4 ASCII
28                      INCL YEAR6 125 4 ASCII
29              STOP
30
31
32              PROC FILLER AT DGRAPHIC AND EGRAPHIC
33              PROC TITLE AT NONE AND ETITLE
34              PROC YEAR1 AT NONE AND EYEAR1
35              PROC YEAR2 AT NONE AND EYEAR2
36              PROC YEAR3 AT NONE AND EYEAR3
37              PROC YEAR4 AT NONE AND EYEAR4
38              PROC YEAR5 AT NONE AND EYEAR5
39              PROC YEAR6 AT NONE AND EYEAR6
40              PROC FILLER AT D1972 AND E1972
41              PROC FILLER AT D1973 AND E1973
42              PROC FILLER AT D1974 AND E1974
43              PROC FILLER AT D1975 AND E1975
44              PROC FILLER AT D1976 AND E1976
45              PROC FILLER AT D1977 AND E1977
46              PROC FILLER AT DHIGH AND EHIGH
47
48
49              REG DATA-VALUE 9(6)
50              REG MAX 9(6)
51
52              REGISTER GRAPHIC X(4) ****
53
54              REGISTER FIELD 9(2)
55
56              REGISTER YEAR-1972 9(6) 5500
57              REGISTER YEAR-1973 9(6) 11000
58              REGISTER YEAR-1974 9(6) 19300
59              REGISTER YEAR-1975 9(6) 25500
60              REGISTER YEAR-1976 9(6) 33900
61              REGISTER YEAR-1977 9(6) 44500
62
63
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
64      *
65      *       THE PROCESSING IS TO STORE THE DATA IN THE PASSING RECORD
66      *
67
68      ETITLE: STORE TITLE
69              RETURN
70
71      DGRAPHIC:
72              DISPLAY GRAPHIC
73              RETURN USING FIELD
74
75      EGRAPHIC:
76              STORE GRAPHIC
77              RETURN
78
79      EYEAR1:
80              STORE YEAR1
81              RETURN
82
83      EYEAR2:
84              STORE YEAR2
85              RETURN
86
87      EYEAR3:
88              STORE YEAR3
89              RETURN
90
91      EYEAR4:
92              STORE YEAR4
93              RETURN
94      EYEAR5:
95              STORE YEAR5
96              RETURN
97
98      EYEAR6:
99              STORE YEAR6
100             RETURN
101
102     D1972:  DISPLAY YEAR-1972
103             RETURN USING FIELD
104
105     E1972:  STORE YEAR-1972
106             MOVE YEAR-1972 TO MAX
107             RETURN
108
109     D1973:  DISPLAY YEAR-1973
110             RETURN USING FIELD
111
112     E1973:  STORE YEAR-1973
113             MOVE YEAR-1973 TO DATA-VALUE
114             PERFORM SETMAX
115             RETURN
116
117     D1974:  DISPLAY YEAR-1974
118             RETURN USING FIELD
119
120     E1974:  STORE YEAR-1974
121             MOVE YEAR-1974 TO DATA-VALUE
122             PERFORM SETMAX
123             RETURN
124
125     D1975:  DISPLAY YEAR-1975
126             RETURN USING FIELD
127
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
128     E1975:  STORE YEAR-1975
129             MOVE YEAR-1975 TO DATA-VALUE
130             PERFORM SETMAX
131             RETURN
132
133     D1976:  DISPLAY YEAR-1976
134             RETURN USING FIELD
135
136     E1976:  STORE  YEAR-1976
137             MOVE YEAR-1976 TO DATA-VALUE
138             PERFORM SETMAX
139             RETURN
140
141     D1977:  DISPLAY YEAR-1977
142             RETURN USING FIELD
143
144     E1977:  STORE YEAR-1977
145             MOVE YEAR-1977 TO DATA-VALUE
146             PERFORM SETMAX
147             RETURN
148
149
150     SUBROUTINE      SETMAX
151
152             COMPARE DATA-VALUE MAX
153             IF LESS ENDSETMAX
154
155     MOVEDATA:
156             MOVE DATA-VALUE TO MAX
157
158     ENDSETMAX:
159             ENDSUB
160
161
162
163     DHIGH:
164             DISPLAY MAX
165             RETURN USING FIELD
166
167     EHIGH:
168             STORE DATA-VALUE
169             COMPARE MAX DATA-VALUE
170             IF GREATER BAD-DATA
171             COMPARE DATA-VALUE "20"
172             IF GREATER LINK-AWAY
173             MOVE "20" TO DATA-VALUE
174
175     LINK-AWAY:
176             MOVE DATA-VALUE TO MAX
177             PASS PASS-REC
178             RETURN
179
180     BAD-DATA:
181             MOVE MAX TO DATA-VALUE
182             GO TO LINK-AWAY
183

****"FINISH" NOT FOUND--INSERTING "FINISH"

 FORMAT LINKED TO BARGRAPH
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
CRAIGS

FIELD      PHYS./LOG.                                    AUTO-  REQ.  FULL  AUTO-
NAME         FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

FILLER         1   1  X(4)          *    *                                  *
TITLE          2   2  X(47)              *
YEAR1          3   3  9(4)              *                    *
YEAR2          4   4  9(4)              *                    *
YEAR3          5   5  9(4)              *                    *
YEAR4          6   6  9(4)              *                    *
YEAR5          7   7  9(4)              *                    *
YEAR6          8   8  9(4)              *                    *
FILLER         9   9  9(6)         *    *
FILLER         10  10 9(6)         *    *
FILLER         11  11 9(6)         *    *
FILLER         12  12 9(6)         *    *
FILLER         13  13 9(6)         *    *
FILLER         14  14 9(6)         *    *
FILLER         15  15 9(6)         *    *
16:29:03 02/03/78
PRINT OF FORMAT: BARGRAPH
@     ZZZ,ZZZ  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  ZZZ,ZZZ




@@    ZZZ,ZZZ  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  ZZZ,ZZZ




@@    ZZZ,ZZZ  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  ZZZ,ZZZ




@     ZZZ,Z29  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  ZZZ,Z29
               XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX        X
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
**********************************************************************************
**********************************************************************************
16:29:03 02/03/78
PRINT OF FORMAT: BARGRAPH
     PHYS./LOG.                                        AUTO-  REQ.   FULL   AUTO-
       FIELD#  DESCRIPTION  DISP  EDIT  OUTPUT  DUPE   ENTRY  FIELD  ENTRY  SEC
        1   1  9(6)          *
        2   2  X(52)         *
        3   3  9(6)          *
        4   4  9(6)          *
        5   5  X(52)         *
        6   6  9(6)          *
        7   7  9(6)          *
        8   8  X(52)         *
        9   9  9(6)          *
        10  10 9(6)          *
        11  11 X(52)         *
        12  12 9(6)          *
        13  13 X(52)         *
        14  14 X(1)               *                                        *
FORMAT NOT LINKED
FIRST LINE USED: 1
LAST LINE USED: 23
AOS SYNTAX REV 01.01      BARGRAPH.VS    BARGRAPH.UF  16:29:41   2/3/78


1       *
2       *
3       *
4       *       ************************************
5       *       *                                  *
6       *       * STAN DURLAND'S BAR GRAPH PROGRAM *
7       *       *                                  *
8       *       ************************************
9       *
10      *
11      *
12      *       THIS PROGRAM IS SUPPOSED TO MAKE THE BAR GRAPH IN THE
13      *       IDEA CONCEPTS MANUAL. USERS ASKED FOR IT, THEY GOT IT.
14      *
15      *
16
17
18
19              NAME    BARGRAPH
20
21              ON END OF DATA QUIT-FORMAT
22
23              REGISTER GRAPH X(52)
24
25              REDESIGNATE GRAPH
26
27                      HEAD 5 47
28                      YEAR1 5 4
29                      YEAR2 13 4
30                      YEAR3 21 4
31                      YEAR4 29 4
32                      YEAR5 37 4
33                      YEAR6 45 4
34              STOP
35
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
36      REGISTER  HEADLINE X(47) DATA GENERAL -- CUMULATIVE COMPUTER SHIPME
37
38      REGISTER ASTERISKS X(4) ****
39
40      REGISTER DASHES X(56) ----------------------------------------↑
41 ------------
42
43      REGISTER SHIPMENTS 9(6) 50000
44
45      REGI Y1 X(4) 1972
46      REGI Y2 X(4) 1973
47      REGI Y3 X(4) 1974
48      REGI Y4 X(4) 1975
49      REGI Y5 X(4) 1976
50      REGI Y6 X(4) 1977
51
52      REGI ST1972 9(6) 5500
53      REGI ST1973 9(6) 11000
54      REGI ST1974 9(6) 19300
55      REGI ST1975 9(6) 25500
56      REGI ST1976 9(6) 33900
57      REGI ST1977 9(6) 44500
58
59      RECD FOR PASSING ACCEPT-YEAR-DATA
60              LEN 129
61              INCL ST1972 1 6 ASCII
62              INCL ST1973 8 6 ASCII
63              INCL ST1974 15 6 ASCII
64              INCL ST1975 22 6 ASCII
65              INCL ST1976 29 6 ASCII
66              INCL ST1977 36 6 ASCII
67              INCL HEADLINE 43 47 ASCII
68              INCL ASTERISKS 90 4 ASCII
69              INCL SHIPMENTS 94 6 ASCII
70              INCL Y1 100 4 ASCII
71              INCL Y2 105 4 ASCII
72              INCL Y3 110 4 ASCII
73              INCL Y4 115 4 ASCII
74              INCL Y5 120 4 ASCII
75              INCL Y6 125 4 ASCII
76      STOP
77
78      REGI S1972 9(6)
79      REGI S1973 9(6)
80      REGI S1974 9(6)
81      REGI S1975 9(6)
82      REGI S1976 9(6)
83      REGI S1977 9(6)
84
85      REG FLOP 9(1) 0
86
87      REGI DELTA 9(5)
88
89      REGI HDELTA 9(5)
90
91      REGI SCROLLKT 9(2)
92
93      REG BLANKS X(52)
94
95      REG TEMP 9(6)
96
97
98
99
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
100     *
101     *       THESE ARE THE PROCESS STATEMENTS. NOTE THAT THERE ARE
102     *       14 FIELDS IN THE FORMAT, SO THERE MUST BE 14 CORRESPONDING
103     *       PROCESS STATEMENTS
104     *
105
106     P1#     PROC    FILLER AT D1 AND NONE
107     P2#     PROC    FILLER AT D2 AND NONE
108     P3#     PROC    FILLER AT D3 AND NONE
109     P4#     PROC    FILLER AT D1 AND NONE
110     P5#     PROC    FILLER AT D2 AND NONE
111     P6#     PROC    FILLER AT D4 AND NONE
112     P7#     PROC    FILLER AT D1 AND NONE
113     P8#     PROC    FILLER AT D2 AND NONE
114     P9#     PROC    FILLER AT D5 AND NONE
115     P10#    PROC    FILLER AT D6 AND NONE
116     P11#    PROC    FILLER AT D7 AND NONE
117     P12#    PROC    FILLER AT D8 AND NONE
118     P13#    PROC    FILLER AT D9 AND NONE
119     P14#    PROC    FILLER AT NONE AND E10
120
121
122
123     *
124     *       NEXT COME THE IFPL LANGUAGE STATEMENTS FOR PROCESSING EACH FIELD
125     *
126
127     D1:     ADD "1" SCROLLKT SCROLLKT
128             PERFORM UPDATE
129             DISPLAY SHIPMENTS
130             RETURN
131
132     D2:     DISPLAY GRAPH
133             RETURN
134
135     D3:     DISPLAY SHIPMENTS
136             GO TO D3A,D3A,D3A,D3A,D3A,D3A USING SCROLLKT
137             MOV "0" SCROLLKT
138             RETURN P4
139
140     D3A:    RETU
141
142     D4:     DISP    SHIPMENTS
143             GO D3A,D3A,D3A,D3A,D3A,D3A USING SCROLLKT
144             MOV     "0"  TO SCROLLKT
145             RETURN TO P7
146
147     D5:     DISP SHIPMENTS
148             GO D3A,D3A,D3A,D3A,D3A,D3A USING SCROLLKT
149             MOV "0" SCROLLKT
150             RETURN TO P10
151
152     D6:     MOV "0" TO SHIPMENTS
153             MOV DASHES TO GRAPH
154             DISPLAY THE SHIPMENTS
155             RETURN
156
157     D7:     DISP GRAPH
158             RETURN
159
160     D8:     DISPLAY SHIPMENTS
161             RETURN
162
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
163     D9:     MOVE BLANKS TO GRAPH
164             MOV Y1 TO YEAR1
165             MOV Y2 TO YEAR2
166             MOV Y3 TO YEAR3
167             MOV Y4 TO YEAR4
168             MOV Y5 TO YEAR5
169             MOV Y6 TO YEAR6
170             DISPLAY GRAPH
171             RETURN
172
173     E10:    LINK USING "BARGRAPHDATA"
174
175
176     QUIT-FORMAT:
177             QUIT
178
179
180     SUBR    UPDATE
181
182             COMPARE DELTA "0"
183             IF NOT-E TAG72
184             ACCEPT ACCEPT-YEAR-DATA
185             DIV SHIPMENTS "20" DELTA
186             ADD DELTA SHIPMENTS TEMP
187             MOV "0" TO SHIPMENTS
188             MOV HEADLINE TO HEAD
189             DIV DELTA "2" HDELTA
190             ADD HDELTA ST1972 S1972
191             ADD HDELTA ST1973 S1973
192             ADD HDELTA ST1974 S1974
193             ADD HDELTA ST1975 S1975
194             ADD HDELTA ST1976 S1976
195             ADD HDELTA ST1977 S1977
196             GO TO END3
197
198
199     TAG72:  MOVE BLANKS GRAPH
200             SUB DELTA TEMP TEMP
201             MOV TEMP SHIPMENTS
202             COMPARE S1972 SHIPMENTS
203             LES TAG73
204             MOV ASTERISKS TO YEAR1
205
206     TAG73:  COM S1973 SHIPMENTS
207             LES TAG74
208             MOV ASTERISKS YEAR2
209
210     TAG74:  COM S1974 SHIPMENTS
211             LES TAG75
212             MOV ASTERISKS YEAR3
213
214     TAG75:  COM S1975 SHIPMENTS
215             LES TAG76
216             MOV ASTERISKS TO YEAR4
217
218     TAG76:  COM S1976 SHIPMENTS
219             LES TAG77
220             MOV ASTERISKS YEAR5
221
222     TAG77:  COM S1977 SHIPMENTS
223             LES END1
224             MOV ASTERISKS YEAR6
225
```

Figure D-8. CRAIGS and BARGRAPH (continued)

```
226      END1:    COM FLOP "0"
227               EQ END2
228               MOV "0" SHIPMENTS
229
230      END2:    SUB FLOP "1" FLOP
231
232      END3:
233               ENDSUB
234
235               FINISH
 FORMAT NOT LINKED

BARGRAPH

FIELD      PHYS./LOG.                                    AUTO- REG.  FULL  AUTO-
NAME         FIELD#  DESCRIPTION DISP EDIT OUTPUT DUPE  ENTRY FIELD ENTRY SEC

FILLER         1   1  9(6)          *
FILLER         2   2  X(52)         *
FILLER         3   3  9(6)          *
FILLER         4   4  9(6)          *
FILLER         5   5  X(52)         *
FILLER         6   6  9(6)          *
FILLER         7   7  9(6)          *
FILLER         8   8  X(52)         *
FILLER         9   9  9(6)          *
FILLER        10  10  9(6)          *
FILLER        11  11  X(52)         *
FILLER        12  12  9(6)          *
FILLER        13  13  X(52)         *
FILLER        14  14  X(1)               *                          *
```

Figure D-8. CRAIGS and BARGRAPH (continued)

End of Appendix

Index

Within this index, the letter "f" means "and the following page"; "ff" means "and the following pages". Also, primary references are listed first.

D

M

N

O

DataGeneral

Software Documentation Remarks Form

How Do You Like This Manual?

Title _____ No. _____

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

- ☐ EDP Manager
- ☐ Senior System Analyst
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ Other _____

What programming language(s) do you use? _____

_____

How Do You Use This Manual?

(List in order: 1 = Primary use)

- _____ Introduction to the product
- _____ Reference
- _____ Tutorial Text
- _____ Operating Guide
- _____ _____

Do You Like The Manual?

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the topic order easy to follow?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name _____ Title _____ Company _____

Address _____ Date _____

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

Data General

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
C/350, C/330, C/300		
S/250, S/230, S/200		
S/130		
AP/130		
CS Series		
N3/D		
Other NOVA		
microNOVA		
Other (Specify) _____		
_____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ SOS
- ☐ RDOS
- ☐ RTOS
- ☐ Other

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ ECLIPSE Cobol
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Interactive
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ RSTCP
- ☐ HASP
- ☐ RJE80
- ☐ SAM
- ☐ CAM
- ☐ 4025
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

Data General

Data General Corporation, Westboro, Massachusetts 01581, (617) 366-8911

FOLD FOLD

STAPLE STAPLE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS · PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator

Southboro, Massachusetts 01772