Data General

Idea Interactive Data Entry/Access Concepts and Facilities (AOS)

069-000023-00

Idea
Interactive
Data Entry/Access
Concepts and Facilities
(AOS)

069-000023-00

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. 069-000023

Revision 00, October 1978

Idea
Interactive
Data Entry/Access
Concepts and Facilities
(AOS)
069-000023

Revision History:

Original Release - October 1978

Preface

This manual describes the capabilities and underlying principles of Data General's Idea system as it runs under the Advanced Operating System, using the INFOS® system file structure. We wrote this manual for both the technical reader who will be implementing Idea systems, and for the general reader who wants an overall description of Idea.

The first five chapters of the manual highlight Idea's general features; chapters six, seven, and eight describe Idea's programming language in detail.

To develop your Idea system, you will need the following manuals and documents:

Idea Product Brief (012-301)
Idea Systems Brief (012-302)
Idea Programmer's Reference Manual (AOS) (093-151)
Idea Release Notice (085-047)
Introduction to the INFOS System (093-113)
The INFOS Storybook (093-199)
INFOS System User's Manual (AOS) (093-152)

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 some 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 of system interactions, we use a right parenthesis,), for the system prompt,

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

End of Preface

Contents

Chapter 4 - Organization of Application Systems

Chapter 5 - INFOS Database Manager

Chapter 6 - IFPL Data Transaction Language--Most Statements

Illustrations

Tables

Chapter 1
Introduction

What is Idea?

Idea stands for Interactive Data Entry/Access. You can use the Idea system to develop and operate database systems that are fully compatible with other Data General ECLIPSE-line software.

The Idea system consists of three major elements -- a language package, a runtime monitor, and a database manager.

The Idea Language Package

Using a system utility called IFMT, you create screen formats that will control data entry and access with your programs. Then you write your applications programs in Idea's Field Processing Language (IFPL). Together, IFPL and IFMT make up the language package.

You can develop your Idea application from the top down. At the top design level, you lay out the database and select the screen formats to interact with it. At the second design level, you design the individual screen formats. The bottom level consists of the IFPL programs that support the various screen formats.

Idea Screen Formats

The fundamental organizational unit of an Idea application is the screen format. This makes the application modular, which makes applications easier to plan and develop. Later, when changes to the system are required, they will probably be isolated to specific formats and their IFPL programs. Similarly, extensions to the system will probably be limited to the addition of new formats and their programs.

Idea screen formats contain their own program control structures, which control such things as processing sequence and inter-screen linking. In its simplest application, a screen steps through its data fields sequentially and, when through, either logs itself off or calls another screen format. You can also modify the behavior of a screen with a supporting IFPL program. Nonetheless, the screen's default control structure is the top-level control structure of the Idea application.

Screen formats consist of data fields plus literal material that does not change in screen execution. In a finished application, the literal material is a guide to the data entry operator; however, during program development, the literal material functions as program comments.

During execution, the system displays these comments on the screen; they permit you to watch the top-level control structure function, as the cursor steps from field to field and from comment to comment. The high visibility of the format control structure thus turns the screen into a powerful design tool.

Screen format data fields contain length and data-type specifications. During the execution, the system checks any data entered in these fields against the specifications; thus the screen format contributes further to the application.

Idea Format Generator --IFMT

To generate screen formats, you use the IFMT utility. Generating screen formats requires no programming; you set up the format on the video display screen just as you want it to appear to someone using it.

IFMT also asks you a series of questions about format parameters; you answer these questions interactively.

IFMT serves three purposes:

1. It defines the overall screen format structure;

2. It defines the individual fields on the format; and

3. It allows you to make screen field definitions available to the compiler. (This is a programming option; you may also describe your fields within your IFPL program.)

The only limit on the number of formats a system may have is set by the amount of available storage.

Idea Field Processing Language -- IFPL

Detail processing of an Idea application is done by conventional programs written in IFPL. Each IFPL program supports a particular screen format.

IFPL programs serve three purposes. First, they provide the interface between the application and the database. Second, they modify the behavior of the screen format, including its control structure. Third, they perform conventional processing through statements that manipulate data, do arithmetic, and provide program control.

An IFPL program has less to do than a traditional applications program because the screen format and the runtime Monitor handle some of the traditional tasks. For example, your IFPL program will not include the screen format. The format is displayed directly from the file created by the format generator.

Furthermore, your IFPL program doesn't have to move the cursor or accept data from the terminal on a character-by-character basis because the monitor handles that function. A single instruction in IFPL stores data input from the terminal. Similarly, a single program statement can display data on the terminal.

The monitor also checks entries for illegal characters, and sends out the resulting error messages. The monitor sends only validated data to your IFPL program.

The Idea Runtime Monitor

Idea applications programs operate under a runtime Monitor. Up to 32 terminals under AOS can operate concurrently, running any combination of the same or different screen formats and programs.

The Monitor is format-driven. That is, the terminal data entry operator can choose and run any one of the available formats totally independently of what may be running on other terminals.

The Monitor is also interactive. It prompts the operator, validates entered data, and flags errors immediately, while the operator is still at the offending field. Its editing functions allow the operator to correct or modify entered data.

Because IDEA applications are interactive and format-driven, you have a great deal of freedom in setting up your data entry and retrieval operations. You can, for example, decentralize data capture terminals, locating them near the data source. Similarly, you can allow operating departments to access data directly, rather than wait for bulky paper reports.

INFOS®, the Database Manager

The INFOS system controls data entry and access to the database. Idea INFOS uses a DBAM (Database Access Method) file structure that is well suited for systems in which many programs share data files.

You access data records via keys through a multilevel indexing structure. This indexing structure allows you to cross-index your records through several indexing paths.

The system posts immediately entered data to the data record, and accessed data is always the latest available. A record locking mechanism assures that new information can't be posted to an open record. These features eliminate delays caused by off-line input, batch processing, and printing and distributing reports.

Concurrent Processing

You can run Idea programs concurrently with other processes, such as programs written in COBOL, RPG II, or other Data General languages. Via the INFOS system, these programs can access the same database and use identical indexing structures.

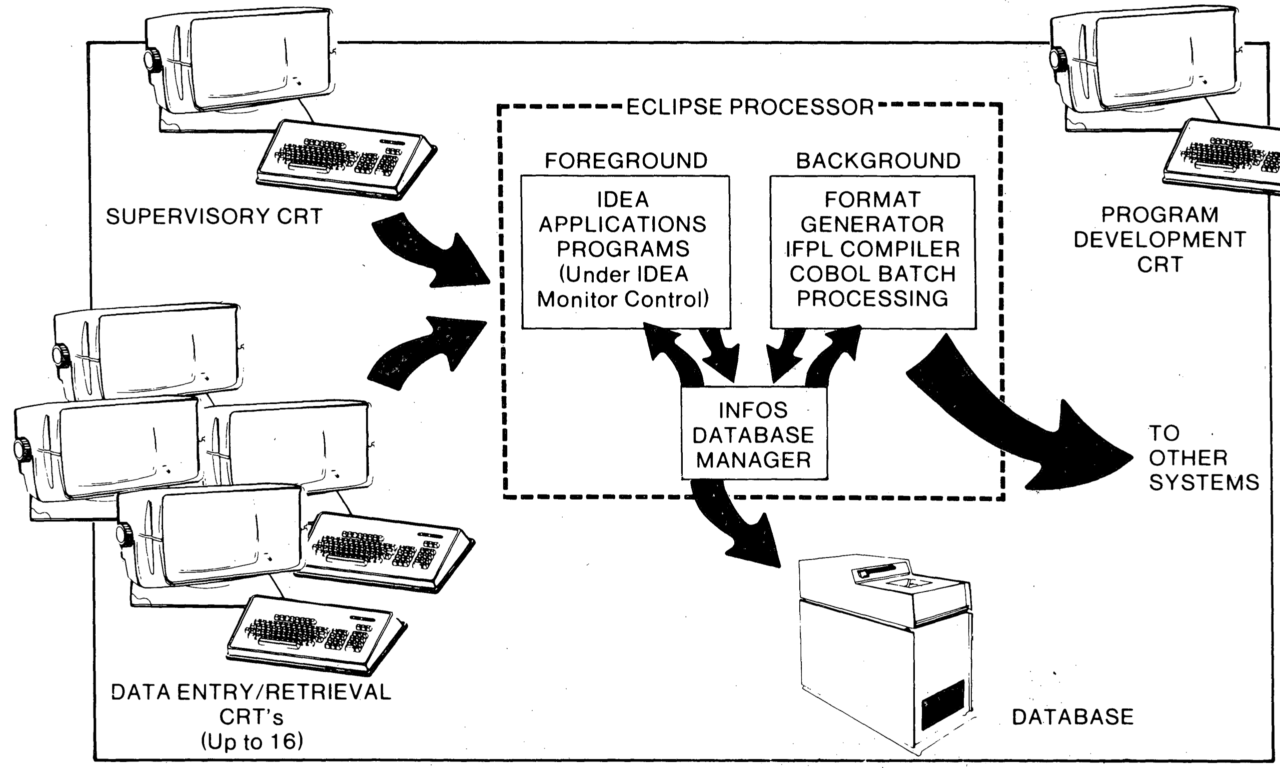
You can also run communications software concurrently with Idea to link your AOS system to other Data General computers or to larger host systems.

Hardware Environment

Under AOS, Idea supports the simultaneous operation of up to 32 operator terminals (video and/or hard copy).

The commercial ECLIPSE computer processor controls the entire system. Among its special features are memory management, an extended arithmetic unit for fast calculation processing, and special business-oriented instructions incorporated directly into its instruction set.

Data General also offers a full range of disk storage devices, which you can link together to provide up to 1536 megabytes of on-line storage. You can use industry-standard magnetic tape formats for on-line back-up of the database, for archival storage, and for inter-system data exchange.


SUPERVISORY CRT
ECLIPSE PROCESSOR
FOREGROUND
BACKGROUND
IDEA APPLICATIONS PROGRAMS (Under IDEA Monitor Control)
FORMAT GENERATOR IFPL COMPILER COBOL BATCH PROCESSING
PROGRAM DEVELOPMENT CRT
INFOS DATABASE MANAGER
TO OTHER SYSTEMS
DATA ENTRY/RETRIEVAL CRT's (Up to 16)
DATABASE

SD-01143

Figure 1-1. The Idea System Provides Full Multiterminal Capabilities

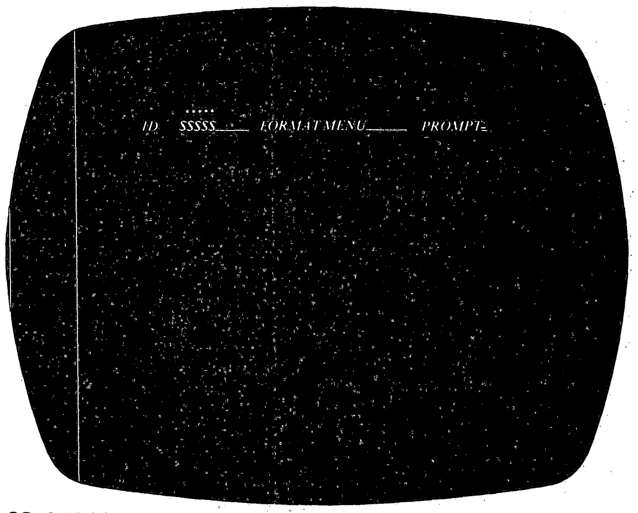
End of Chapter

Chapter 2 Operation

Idea can support up to 32 data entry/access terminals operating concurrently under AOS. The terminals can run any combination of the same or different formats and their associated applications programs. You can use the sign-on function to call up a format and its program from the system.

Accessing Formats-The Sign-On Function

Sign-on provides two primary functions: convenient access to all system formats and security from unauthorized use. Here is how it works.



```
      *****
ID    SSSSS____   FORMAT MENU_____   PROMPT=
```

SD-01144

Figure 2-1. Idea Sign-on

You can call up a format from any unused terminal in the system. After you log on, the terminal will respond with the log-on display (shown in Figure 2-1). You then type an identification code or a password and the name of the format you want to use. The system will respond by displaying the format; it will also position the cursor on the first field that you must fill out.

You must implement password handling in your applications program. The ID you use to call a format is available to your applications program in the reserved word PASSWORD. The terminal number is available in the reserved word CRT. Your applications program can use these words to implement a variety of different security systems, as required by the sensitivity of the data being handled.

In addition, at start-up time the system operator may designate a particular format (e.g., the menu) as the ground state format. If a format has been previously designated, the terminal operator no longer can choose a format directly at sign-on or at any other time. Instead, the system will display the ground state format, which links to other formats in the system. You may make the links conditional on the password and CRT number.

Data Entry

You enter data into an Idea field via the terminal keyboard. Using the various editing function keys, you can position the cursor anywhere within a field and correct and keyboard errors you've made. After completing a field entry, strike the NEW LINE or RETURN key.

The Idea Monitor will examine each field entry. If it discovers an error, it will display a message and position the cursor at the offending field. If the data meets all its defined criteria, the Monitor will make the data available to the applications program. If the data field is an output field, the data is also output to the transaction file. The Monitor will then position the cursor at the next field.

When you've completed the data entry for a format, the system will flush the format from the screen. It will then display the next logical format (if one is defined), or it will ask you to choose another format.

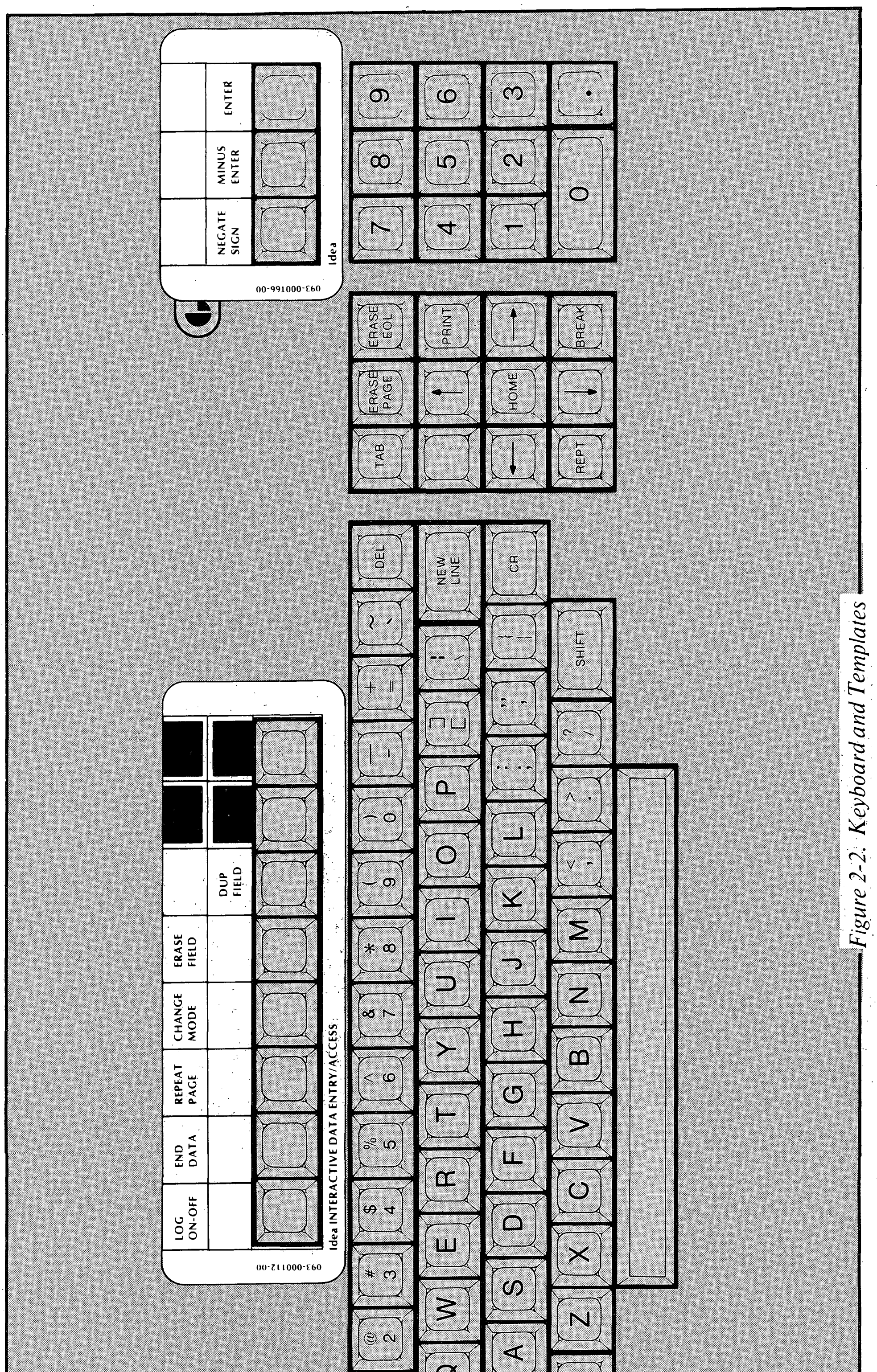


Figure 2-2. Keyboard and Templates

Operator Commands

You can modify the usual sequence of operations described above by issuing keyboard commands via the various keyboard function keys.

To re-edit a field, press backtab; it will move the cursor back to the previous data field.

To end the present format and display the next format (if one has been defined), press End Data.

To end use of the terminal and make it available to another user, press Log Off.

To cancel the present format and start it afresh, press Repeat Page.

To end processing of scroll fields (see below) and move the cursor to the first field beyond the scroll area, press Change Mode.

To make corrections, press Erase Field. It will delete the entry and position the cursor to the field's first character position.

To duplicate the data in the field above (in scroll mode) use Dup Field.

Function Keys

We have implemented the command keys just discussed as part of the Idea system. In addition, you can define five other key functions with your applications programs. Using the last two keys in the function key row (see Figure 2-2) by themselves, and with the Shift Key, you can define four functions; the Escape key provides another.

With suitable code in your applications program, you can set commonly-used operator functions to simple keystrokes. Examples of possible functions are:

- Escape. In applications systems incorporating several levels of menus, you can use the Escape key to back up to the preceding menu.
- Skip. You could use a Skip key to bypass an inapplicable section of a format.
- Subtotal. You could display the subtotal for a column of scrolled entries at any point in the entry process.

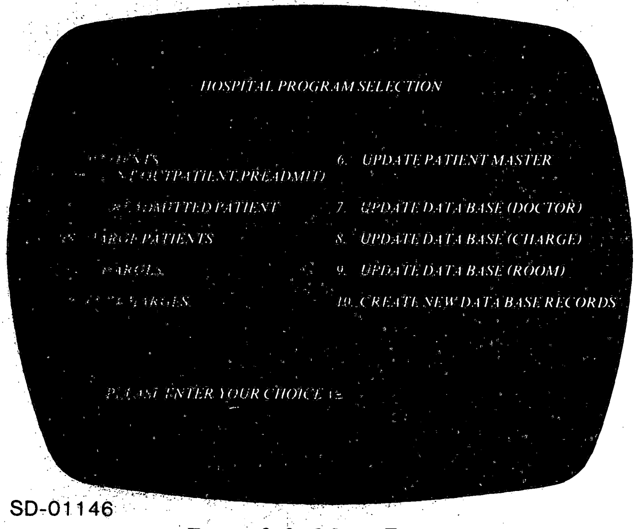
Examples of Formats

The examples below show the kind of format structures and operations that you can create on the Idea system.

Menu

The menu is a clear way to present an operator with all the options available at a particular processing step. Its clarity makes the system more natural to operate and means that less training is required.

For instance, you can use the menu as the system ground state format to provide access to all other formats in the system. Then, at sign-on, the menu format is the only one the operator need remember. You can implement the menu format with a simple and short applications program with as few as nine IFPL statements.


HOSPITAL PROGRAM SELECTION
OUTPATIENT,PREADMIT)
ADMITTED PATIENT
PATIENTS
6. UPDATE PATIENT MASTER
7. UPDATE DATA BASE (DOCTOR)
8. UPDATE DATA BASE (CHARGE)
9. UPDATE DATA BASE (ROOM)
10. CREATE NEW DATA BASE RECORDS
ENTER YOUR CHOICE

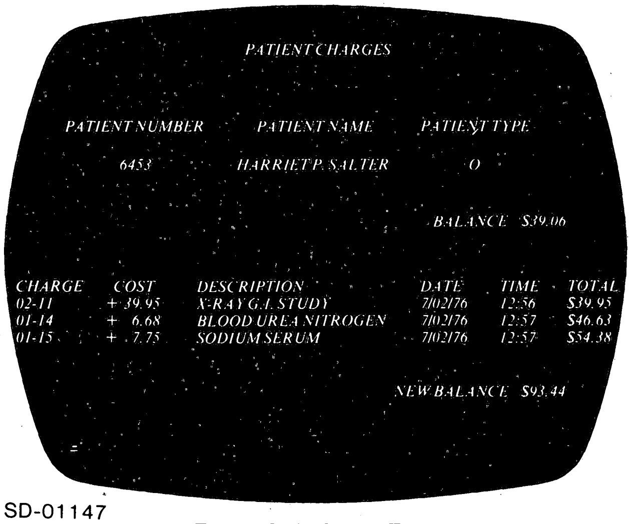
SD-01146

Figure 2-3. Menu Format

Scroll Fields

You will find scroll fields convenient for entering tabular material. For instance, the patient charges list in Figure 2-4 is a scroll field. After you have entered data on the last line of the field, the charges scroll up one line, vacating a line for the next entry.

A screen format can have several independent scroll fields. Scroll fields can total up to 512 bytes and occupy from one line to all twenty-three lines of the screen. The literal heading information is always visible; it does not scroll.



```
                        PATIENT CHARGES

  PATIENT NUMBER      PATIENT NAME        PATIENT TYPE

       6453          HARRIET P. SALTER         O

                                         BALANCE  $39.06

CHARGE    COST     DESCRIPTION             DATE     TIME    TOTAL
02-11    + 39.95   X-RAY G.I. STUDY        7/02/76  12:56   $39.95
01-14    +  6.68   BLOOD UREA NITROGEN     7/02/76  12:57   $46.63
01-15    +  7.75   SODIUM SERUM            7/02/76  12:57   $54.38

                                     NEW BALANCE  $93.44

  =
```

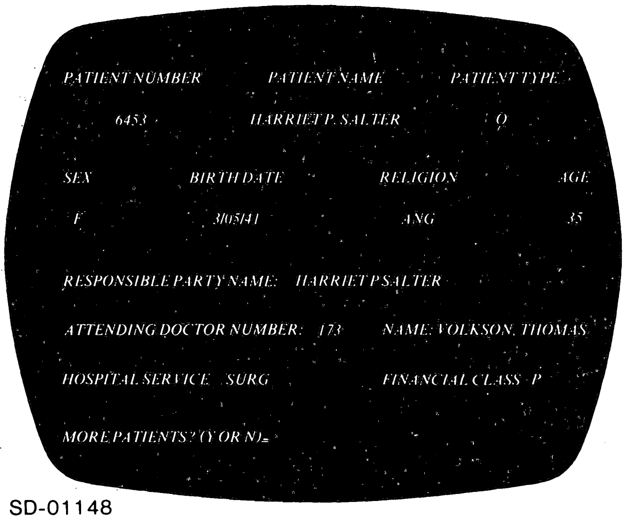
SD-01147

Figure 2-4. Scroll Fields

Enter and Calculate

The scroll format shown in Figure 2-4 illustrates another technique useful in format design. Here, the operator enters only enough information on each of the charges to completely specify it. In our example, the department and charge number specify a change. The program then uses these to look up the cost and the description. From these it calculates the new total and displays it immediately in its field and at the bottom of the screen.

This scheme means that you save operator time and eliminate calculation errors; and the feedback information allows the operator to confirm the input.



```
PATIENT NUMBER          PATIENT NAME          PATIENT TYPE

     6453             HARRIET P. SALTER            O

SEX            BIRTH DATE            RELIGION            AGE

 F              3/05/41                 ANG               35

RESPONSIBLE PARTY NAME:   HARRIET P SALTER

ATTENDING DOCTOR NUMBER:  173        NAME: VOLKSON, THOMAS

HOSPITAL SERVICE   SURG              FINANCIAL CLASS  P

MORE PATIENTS? (Y OR N)_
```

SD-01148

Figure 2-5. Lookup and Update

Lookup and Update

Another convenient format structure to use is the look-up and update format (shown in Figure 2-5). You can use this format for inquiry or to update information. In a typical case, the operator enters data identifying the record needed, such as the name or number of a particular patient. The screen will display data on the patient, as currently reflected. The operator may then use this data for reference or update it with new information.

Screen Overlays

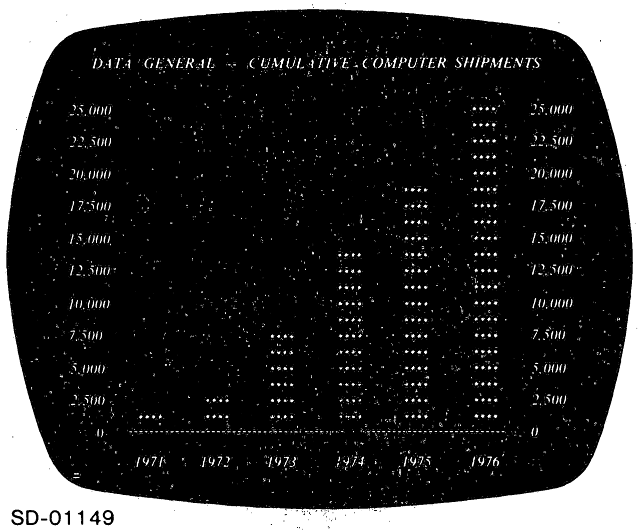
In some applications, you can enter different kinds of data about the same subject. Therefore, while a portion of the format may change, its heading remains the same. In this kind of situation, the overlay is a convenient function. For example, the formats shown in Figures 2-4 and 2-5 share the same patient name and number and could utilize the overlay function. Let's look at how to use overlays.

The system displays the first format, one portion of which contains heading data. Another portion contains data on the first task. After you have filled out the format, it links to the format for the second task, a partial screen format. This second format overwrites a portion of the first format's data but leaves the heading visible. Thus you do not need the data again, nor do you need to look it up in the database.

The overlay feature is quite versatile. You can divide the screen in any proportions because an overlay format can consist of any number of disconnected parts. Parts can start and end anywhere, including in the middle of lines. Furthermore, you can use as many overlay formats as you desire, although only one format can be active at any given time.

Bar Graphs

Some data is best understood when presented visually (see Figure 2-6). The system can generate graphs directly from data stored on the database or from data input to a format. The up-to-the-minute currency of the information, coupled with its visual presentation, make this a powerful display in management information system applications.

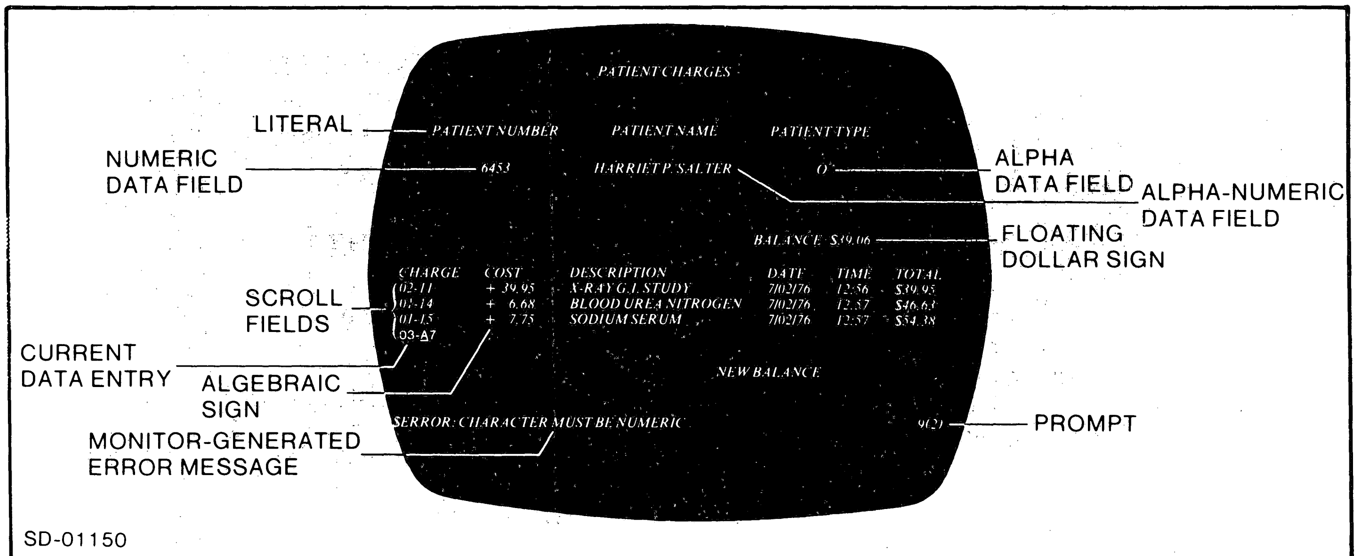


SD-01149

Figure 2-6. Bar Graph Format

End of Chapter

Chapter 3
Format Structure and Preparation



```
                    PATIENT CHARGES

PATIENT NUMBER      PATIENT NAME          PATIENT TYPE
          6453      HARRIET P. SALTER          O

                                   BALANCE  $39.06

CHARGE   COST     DESCRIPTION              DATE     TIME   TOTAL
02-11    + 39.95  X-RAY G.I. STUDY         7/02/76  12:56  $39.95
01-14    +  6.68  BLOOD UREA NITROGEN      7/02/76  12:57  $46.63
01-15    +  7.75  SODIUM SERUM             7/02/76  12:57  $54.38
03-A7

                              NEW BALANCE

$ERROR: CHARACTER MUST BE NUMERIC                           9(2)
```

Figure 3-1. A Format During Data Entry

A format consists of a protected literal area, one or more data fields and a message field.

Literals

You use literals for the format title, to identify data fields, and perhaps for brief instructions to the operator. Literals are protected; you cannot change them from the operator's terminal keyboard. All keyboard characters are legal as literals.

Data Fields

You use data fields for keyboard entry and for data display. You give each field a defined format and length which also specifies the class of characters which are legal for the field. You can restrict the field to to numeric only, alpha only or alphanumeric characters. For numeric fields, the definition includes the decimal point location. It can specify that leading zeros be suppressed; and it can include a floating arithmetic sign or a floating currency symbol.

Data Field Attributes

You assign each data field one of more of the following attributes: Edit, Display, Output, Full, Required, Auto-Dup, and Auto-Entry. Each allows you to control data entry, display and storage for the field in a specific way.

Edit

You use an edit field to enter data into the applications program.

Display

The format displays data generated by the applications program in this field.

You can assign both Display and Edit attributes to a field. In this case, it is a display field the first time it is encountered and an edit field thereafter. While a field is a display field, it is protected from keyboard entry.

Output

The system automatically stores data entered into the field in the transaction file. You can assign the attribute Output to Display or Edit fields. The data may originate from the applications program (Display and Output), or it may originate from the keyboard (Edit and Output or Output only).

You can assign the following attributes to Edit or Output fields; you cannot use them with Display-only fields.

Full.

You must fill out a Full field completely or not at all. Use this attribute for fields that have a constant number of characters, such as a zip code or a telephone number.

Required.

The operator cannot bypass a required field. Once you encounter it, you must make an entry in it. In addition, the system bars the operator from exiting the format with the Log Off or End of Data commands until all required fields are filled. You can also give a required field the Full attribute.

Auto-Dup.

To save keying effort, you give a field the Auto-Dup attribute. This means that you do not have to retype information each time you encounter a field whose data repeats on every line in a scroll area. You do not have to strike New-Line or Return to terminate this type of field. You type it once and it is automatically duplicated after that.

The Auto-Dup function is useful for output fields where each line of the scroll area forms a separate data record. Auto-Dup inserts a repeating variable into each record without requiring it to be manually re-entered each time.

Secure.

Secure fields echo asterisks during and after operator input. You can use this attribute to protect the privacy of input data.

Auto-Entry.

Once you have filled all character positions in an Auto-Entry field, the system performs the Enter function automatically.

Message Field

The system displays error and information messages (generated by the Monitor or an applications program) on the bottom line of the screen.

Creating Formats

You create formats interactively, under the utility IFMT (pronounced I-format). Here is how to prepare the format.

After placing the terminal in Literal mode, you type the literal data just as it is to appear to the operator. The editing functions allow you to correct errors and move the literals on the screen by inserting and deleting characters and lines.

You can create the data fields by changing the terminal to Field mode. Then type the data fields in place just as you do literals. You can specify the class of characters legal for each field, the length of the field, the position of the decimal point, a currency symbol, an algebraic sign, and commas.

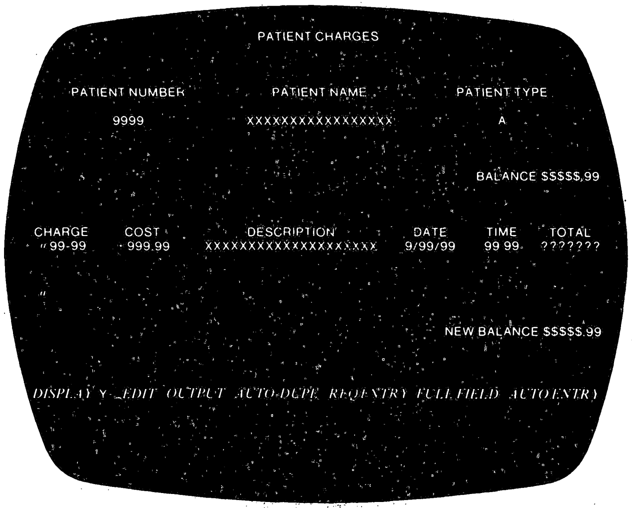
An IFMT keyboard command prints out the screen format. The printout is a character-by-character image of the format, showing the exact placement of all the elements on the screen. The printout indicates data fields by the same mnemonics you used to specify them.

You can switch from Literal mode to Field mode and back any number of times. You may also move the elements of the format and change them repeatedly. When they are just as you want them, put the terminal into Assign Attributes mode, which allows you to assign each data field its attributes.

The format is now complete, so the system will flush it from the screen and store it in the database. From there, it is available to the runtime system. It is also available to IFMT for further revisions.

Hardcopy Printout Formats

IFMT also lets you generate formats for printing reports. In printout mode, IFMT allows formats which are any number of lines long. In the other modes, the print and display formats are identical. For all modes, however, you use the same kind of structures. Thus, reports consist of data fields and literals as you have set up the elements interactively in the screen--a much easier process than through programming. Similarly, you can type headings and other literals directly, rather than in a program.



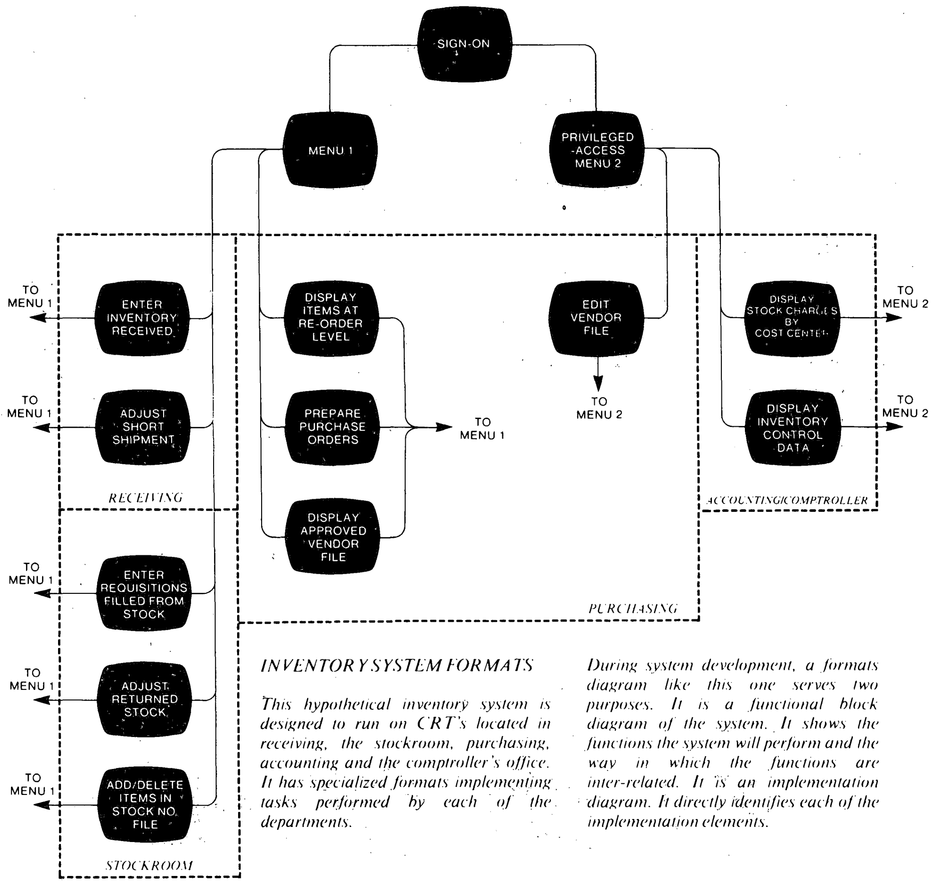
```
                          PATIENT CHARGES

   PATIENT NUMBER           PATIENT NAME            PATIENT TYPE
        9999             XXXXXXXXXXXXXXXXX                A

                                                 BALANCE $$$$$.99

CHARGE     COST          DESCRIPTION           DATE     TIME    TOTAL
"99-99    999.99    XXXXXXXXXXXXXXXXXXXX     9/99/99   99 99   ???????

"
                                             NEW BALANCE $$$$$.99

DISPLAY Y EDIT OUTPUT AUTO-DUPE REQENTRY FULL FIELD AUTOENTRY
```

SD-01151

Figure 3-2. Format in Preparation

End of Chapter

Chapter 4
Organization of Application Systems



INVENTORY SYSTEM FORMATS

This hypothetical inventory system is designed to run on CRT's located in receiving, the stockroom, purchasing, accounting and the comptroller's office. It has specialized formats implementing tasks performed by each of the departments.

During system development, a formats diagram like this one serves two purposes. It is a functional block diagram of the system. It shows the functions the system will perform and the way in which the functions are inter-related. It is an implementation diagram. It directly identifies each of the implementation elements.

SD-01152

Figure 4-1. A Sample Idea Application: An Inventory System

An applications system consists of all the formats, file structures and applications programs required to perform a function; e.g., inventory control, accounts receivable, or hospital patient records.

Formats, Idea's basic organizational units, correspond to data transactions. For example, the inventory system diagrammed in Figure 4-1 has formats for transactions of several types.

There are data entry formats for inventory received and withdrawn, as well as special formats for adjusting errors and reversals. There are also formats for reviewing and changing the system's semi-permanent parameters such as the approved vendors list and re-order levels. There are inquiry formats ranging from specific ("How many units of item X are on hand?") to highly summarized (for business information). In short, there is a format tailored to every transaction and inquiry function we want the the system to fulfill.

Formats and Applications Programs

You can support formats with applications programs to add considerable power to the system. Your programs can perform calculations on data, retrieve data from the database, and display data on the CRT, among many other things.

An applications program is permanently associated with particular format. When you call a format (at sign-on, for instance), the Monitor will automatically load its associated program.

You write applications programs in IFPL, an acronym for Idea Field Processing Language. The structure of IFPL programs suggests the name; that is, each program's structure closely parallels the structure of the format it supports. A specific procedure in its supporting program processes each format data field. The procedures and data fields are brought together in the system at compile time.

During processing, the Monitor processes the format field-by-field. At each field, it gives the applications program control, for a specific procedure. When processing of the procedure is complete, the program passes control back to the Monitor. Because the Monitor takes care of the overhead involved with terminal communication and keeps track of the current and next field, procedures in the applications program are short and direct. They must provide only data processing instructions, not communications-related ones. An example of a format and its procedures is shown in Figure 4-2.

Database. While the format-program modules are specialized to each task, the database is common to all. All programs can access the same records through the identical indexing structure. For more information on the database structure, see Chapter 5.

Monitor Functions

In addition to providing an interface between the format and the program, the Monitor performs several other functions which reduce the burden on your programs. These include error checking, creation of a file to accomodate the data, and the linking of sequential formats.

Error Checking

The Monitor automatically checks for errors in entered data by comparing it to the criteria established during the format creation process. That is, it checks the number of characters, class of characters (alpha, numeric or alphanumeric) and legal positioning of the decimal point in decimal numbers. If the Monitor detects an error, you must correct it before any further data is accepted.

Transaction File

The transaction file is a file in which the Monitor automatically stores data from selected screen fields. You can use the file as a journal or audit trail, or you can use it to transfer data to other systems.

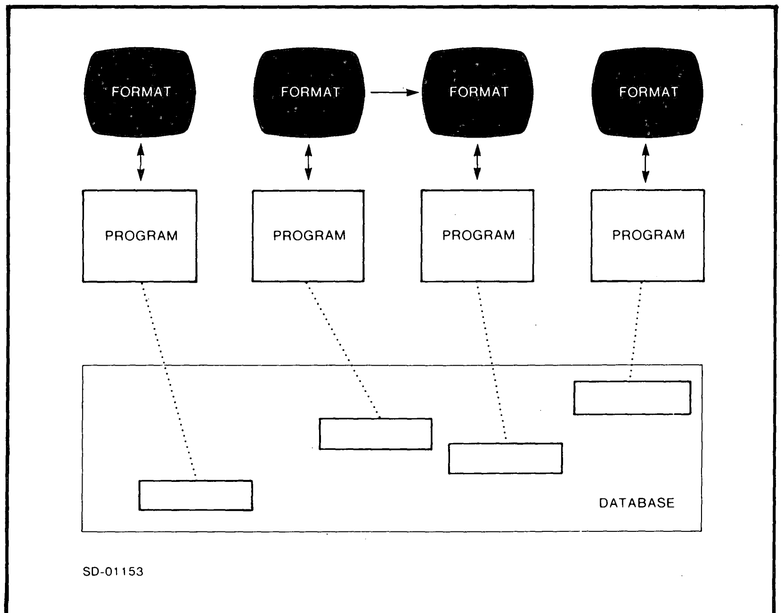


Figure 4-2. A Format and its Procedures

The Monitor automatically creates records tailored to the data input requirements of each format. As you enter data into the format, the Monitor will automatically file it in the database. The Monitor indexes the data records by CRT number, batch number, and format type. It also records date, time and operator ID.

IFPL programs explicitly store and read data in your files. The transaction file is additional to these.

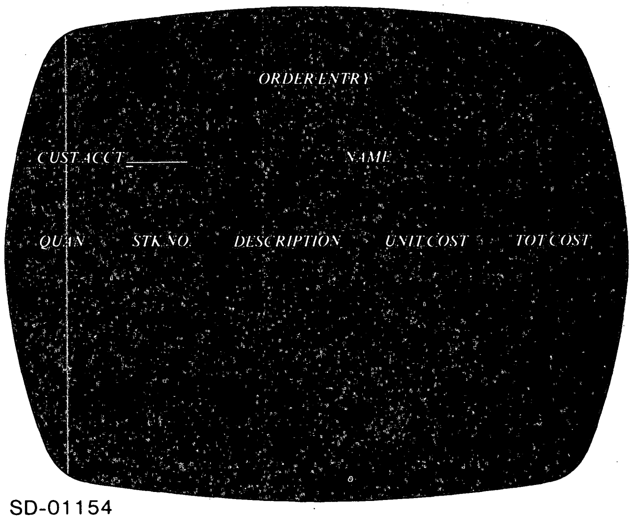
Linking Formats

Sometimes it is convenient to implement a particular task using two or more formats. To do this, you can link formats, so that when one format is completed the system automatically displays the next format required for the task.

The system maintains a facility for passing information from the program of one format to the program of another.

Format-Only Systems

The Monitor performs the above functions--error checking, transaction file entry, and format linking--entirely from parameters you specify when you generate each format. It derives criteria for error checking from the field definitions, it creates the transaction file records directly from the data fields, and it links formats as specified by the format designer. Thus, these functions require no program support. In fact, you can run simple data entry formats without any program support. The Monitor will check data you enter into formats of this type and, after validation, stores it in the transaction file.



SD-01154

Figure 4-3. Sales Order Format

IFPL/Format Programming Example

To understand the field orientation of IFPL and the way in which programs are written, consider this short example (illustrated by Figure 4-3). Each of the fields has a procedure in the program.

For instance, the procedure for the first field (CUST ACCT) stores the account number keyed in by the operator in a variable location called ACCT, as follows:

```
CUSTACCT:  STORE ACCT
           RETURN
```

A single instruction is all that you need to read the account number from the screen and store it. The RETURN returns control to the Monitor, which then positions the cursor to the next field.

The next field, NAME, is a display field. The program displays the customer name associated with the account number so the operator can verify it. The procedure is as follows:

```
CUSTNAME:  FIND CUSTREC USING ACCT
           DISPLAY NAME
           RETURN
```

Using ACCT as the key, our program finds the customer record in the database. The screen then displays the customer's name, a field within the record. Again, the program returns to the Monitor.

The next fields are Quantity and Stock Number; the procedures are similiar to the CUSTACCT procedure above.

```
QUAN:      STORE QUANTITY
           RETURN
STKNO:     STORE STKNO
           RETURN
```

Our program next looks up and displays the Description and Unit Price of the item in question. These procedures are similiar to the CUSTNAME procedure above. In this case STKREC is the item's record. It contains the fields DESCRIPTION and UNITPRICE.

```
DESCS:     FIND STKREC USING STKNO
           DISPLAY DESCRIPTION
           RETURN
PRICE:     DISPLAY UNITPRICE
           RETURN
```

The last field extends the price for item, derived by a calculation, as follows:

```
EXTEND:     MULTIPLY QUAN UNITPRICE TOTPRICE
            DISPLAY TOTPRICE
            RETURN
```

This is the entire procedure section for the format shown. (In addition, however, there is also a definition section; see Chapters 6 and 7 for information on this.)

To summarize, the program shown has accepted operator data from the screen, used it to look-up and display additional data and to make calculations. The program is modular and each module supports a specific data field. Consequently, it is an easy program to specify, write, and modify.

Printing

Interactive systems frequently require two types of printing facilities-- one for the traditional high-volume generation of reports and another for preparing hardcopy for immediate reference. Typically, report printing produces a number of long reports at scheduled intervals. On the other hand, reference printing is used to produce short printouts -- perhaps of a single transaction -- on demand. Idea provides facilities for both report and reference printing.

Report Printing -- PRINTF and COBOL

PRINTF is an Idea utility for printing reports from IDEA-generated data. It is designed to transfer the image of a video terminal display to a line printer. In practice, it is a versatile tool that can generate a variety of transaction or summary reports.

You may also use COBOL to generate conventional reports from the same database.

Report Printing -- DASHER

To generate printed reports, you can attach a DASHER printing terminal to the system as an Idea terminal, have it monitor the other terminals, and have it print at their request. The Idea applications language provides a facility that permits such a terminal to remain quiescent until it senses a print request. (See "Inactivity Clause" in Chapter 6.)

This particular configuration is a convenient way to provide local printing for remote sites. The DASHER can be connected via communications lines in the same way as any other terminal.

Reference Printing -- DASHER

You can also use the DASHER as a reference printer, working as a satellite of a particular video terminal. With this configuration, the DASHER can print an image of its associated video terminal screen.

A typical use for this facility is in an inquiry system that demands a file search using approximate keys. When the operator has retrieved the wanted file, he/she can produce an immediate snapshot of it; the system will ignore the rest of the terminal transaction.

End of Chapter

Chapter 5
INFOS Database Manager

The Idea system uses Data General's powerful INFOS data manager, as implemented at its highest level, DBAM (Database Access Method).

Under the INFOS system, a database is truly just that--a pool of data accessible to various programs for their own purposes. For example, consider a database containing sales orders. Under the INFOS system, we can naturally and conveniently use the same database for billing, for calculating sales commissions, and for statistical management reports, to name just three applications.

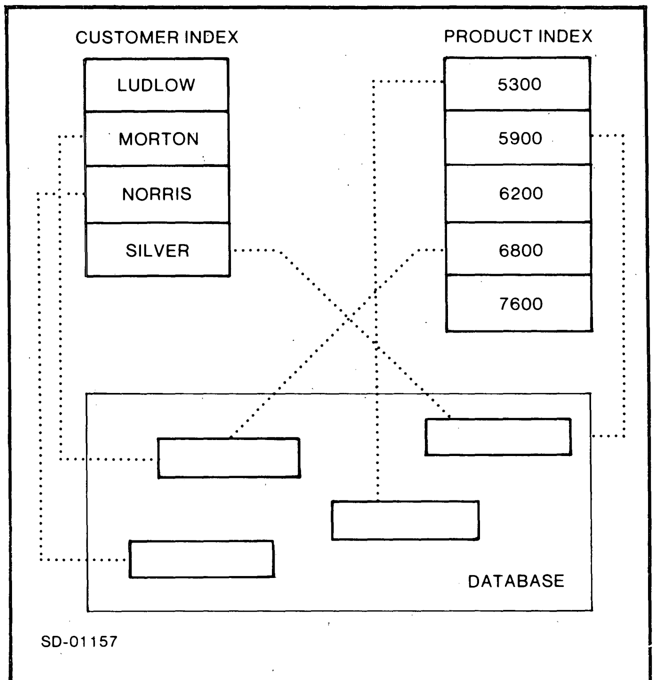


Figure 5-1. Accessing Records via Two Indexes

You do not have to write the programs accessing the database all in the same language, nor operate all of them under Idea. Thus data entry in our example could be under Idea control. The management report program could be one of the IFPL programs comprising an interactive, on-line management inquiry system. On the other hand, you might write the billing and sales commission programs in COBOL, RPGII or another language and run them in batch, concurrently. However, although this may be significant in many cases, it may not be the most important benefit of the INFOS system. Under the INFOS system, the database becomes a growing corporate resource, ready for use in the future in ways you did not envision when you collected the data.

INFOS Structure

The ability to access the same database differently for different purposes is inherent in the way the INFOS system structures data. The database consists primarily of records and indexes. You need to store each record only once, but you may access it through any number of indexes by attributes, thus making it available for different purposes. To continue our example, the billing program could access sales order records through a customer index, while the management report program might access them through a product index as shown in Figure 5-1.

As shown in Figure 5-2, an indexing structure may have an index and several levels of subindexes. Thus, you can easily create hierarchical data structures. For instance, you might organize sales records by year, month, and account. In such a structure, year keys (entries in the year index) point to month subindexes.

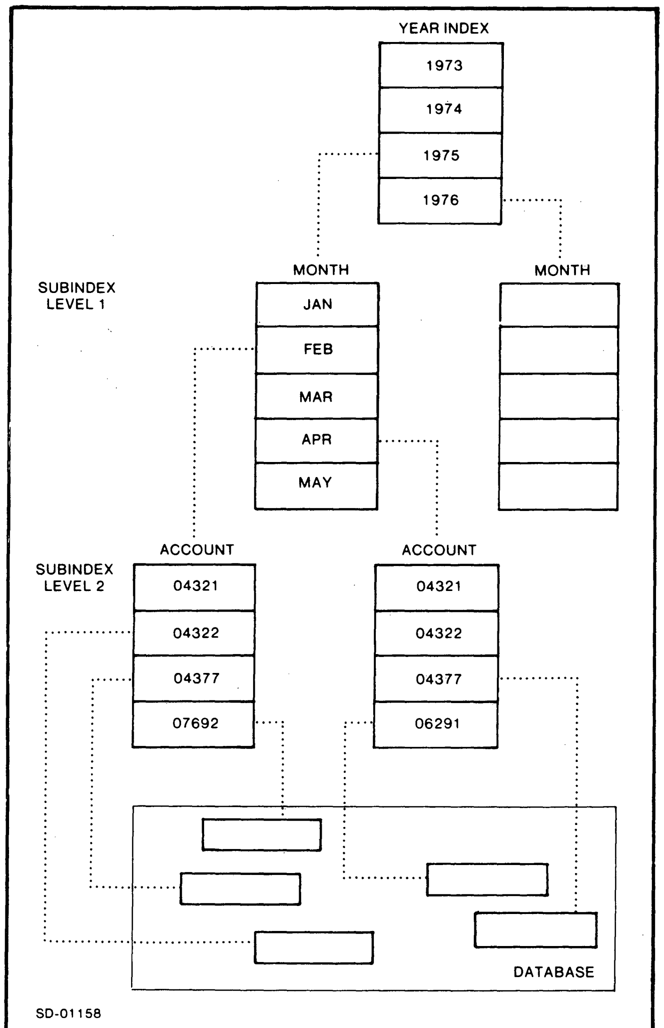


Figure 5-2. Multilevel Indexes

Month keys point to account subindexes. Keys in the lowest subindex point to individual records. In that way, July 1973 is distinguishable from July 1974.

You may note that the number of indexes and subindexes is not limited by the INFOS system. However, each IFPL applications program can open no more than three files (indexes), sharing a total of 15 indexes and subindexes. However, you can apportion them in any way. At one extreme, your IFPL program might handle a single file with an index and 14 levels of subindexing. The transaction file and the common file and their indexes and subindexes are not included in this limit because they are automatically available.

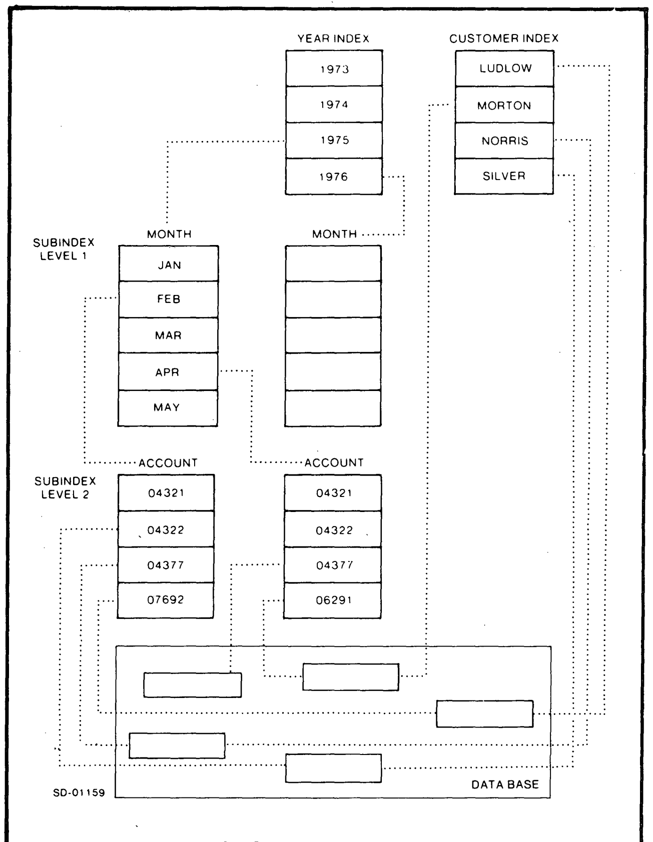


Figure 5-3. Separate Index Paths with Different Numbers of Levels

Of course, you do not have to give all indexing paths the same number of levels. As shown in Figure 5-3, you might access records along one path through only a single index and along another path through a dozen levels. In fact, the INFOS system is quite flexible in the types of index structures and records that you can use and allows many variations of the basic index structure. Each extends the usefulness and convenience of the database.

For instance an index or subindex may reference subordinate subindex levels as well as records directly (see Figure 5-4). That is, you might access sales orders via the sales representative - month, path, and expense reports directly via the sales rep index.

Furthermore, any number of keys within a single index can point to the same record or subindex (see Figure 5-5).

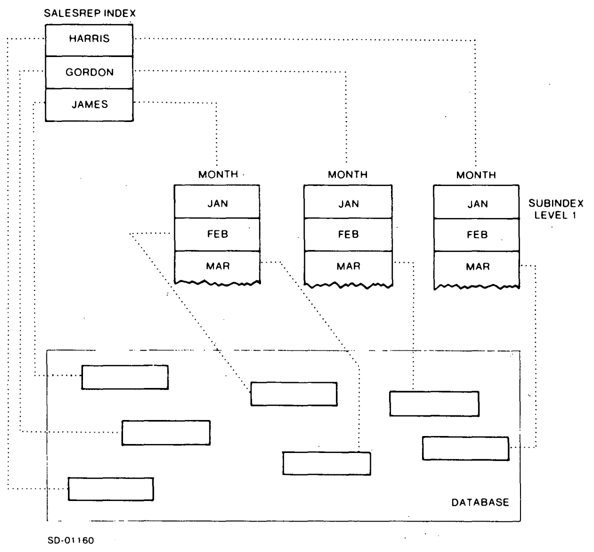


Figure 5-4. You Can Access Records through Index Keys Directly or through Subordinate Subindex Levels

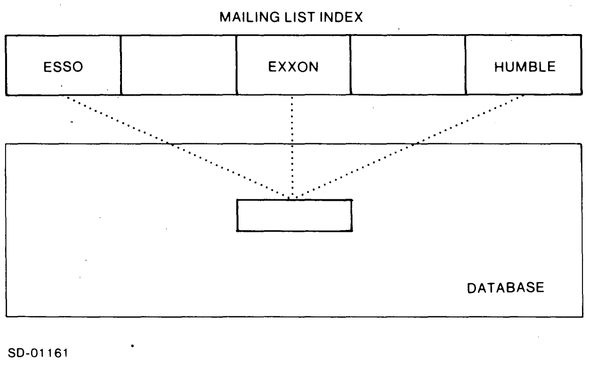


Figure 5-5. A Number of Keys within an Index Pointing to the Same Record

Figure 5-6 shows how the INFOS system supports duplicate keys at all levels. The system distinguishes them from each other by an automatically-generated occurrence count which is permanently associated with the key. Thus SMITH occurrence 20 retains its unique identity even after SMITH occurrence 19 has been deleted from the database. New SMITH added to the database are given a higher occurrence count. Occurrence counts used by deleted keys are not reassigned.

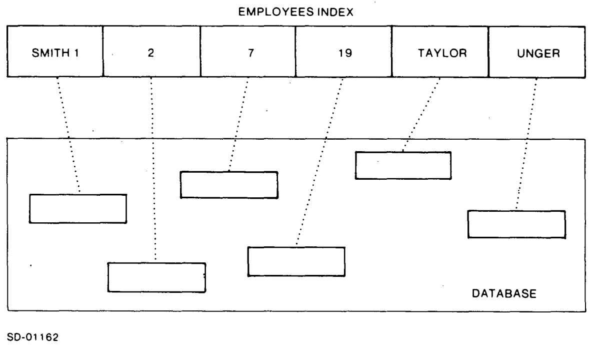


Figure 5-6. Use of Duplicate Keys

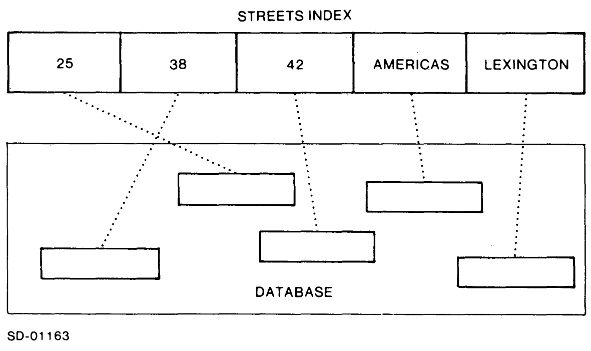


Figure 5-7. Keys of Different Data Type and Length within a Single Index

Within an index, you can use keys of mixed type and of different lengths (see Figure 5-7). The system also supports access via full keys, generic (also known as partial) keys and approximate keys (see Figure 5-8).

The approximate key "SMIDT" will access the record stored under "SMITH." The generic key "SMIDT" will return nothing, since you must give an exact match of the generic key fragment.

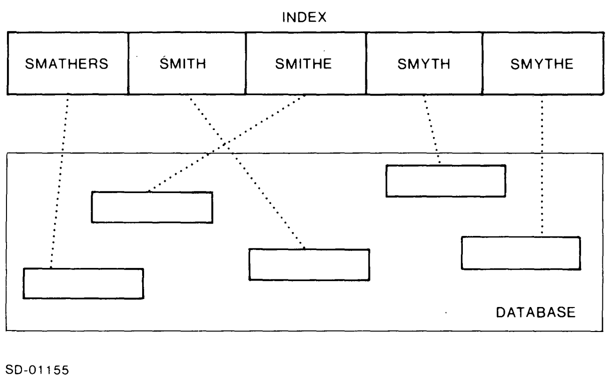


Figure 5-8. Use of Approximate and Generic Keys

You can also access records of different length, which contain different types of information, via the same index path. Futhermore, you can lock records on an individual basis and prevent simultaneous access to the same record by different, unsynchronized users. However, all users can have simultaneous access to all other parts of the same files.

Backing Up the Database

No database system can be considered fully secure unless a good method for recovery from database malfunctions exists. In Idea, the heart of the recovery procedure is the tape logging function (which is completely under program control). All records written to the disk can also be written to tape to provide a backup copy. As further explained in Chapter 7, the logging records are defined and written to tape by your program. Thus, the system designer can implement any of a wide range of schemes.

If you use two tape drives, the system will switch to the next drive automatically, so that terminal operations can continue without interruption for tape mounting and unmounting.

Figure 5-9 illustrates one backing scheme in which IFPL programs write after-images of all records written, re-written, and deleted. In the event of a database crash, a COBOL utility program will update the most recent valid disk image using the after-image.

Creating Database Structures

You can create, modify and delete the database structures described above through a series of interactive INFOS utilities. From the answers you give, the utilities will create or modify a database, requiring no programming. However, designing a large database to use storage efficiently and to have good access time does require specialized knowledge.

The INFOS utilities commonly used in database maintenance are: ICREATE, ICOPY, IDELETE, IRENAME, and INQUIRE.

The utilities are described in the "Utilities" chapter of the *INFOS System User's Manual (AOS)*.

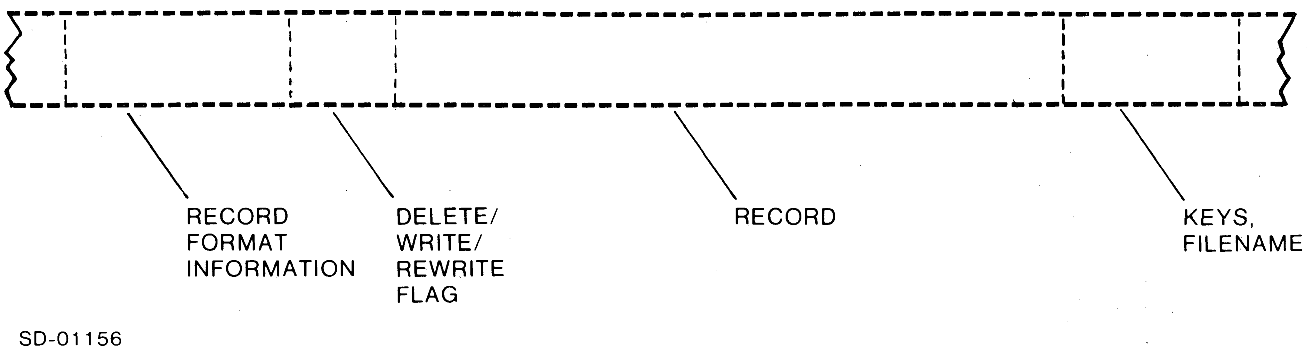


Figure 5-9. Backing up the Database

End of Chapter

Chapter 6
IFPL Data Transaction Language--Most Statements

IFPL is a conversational computer language designed specifically for writing data transaction programs. IFPL operators are English words such as COMPARE, DISPLAY, STORE, and ADD. You can include auxiliary words like TO, FOR, and IS within statements to make programs more readable.

We describe the IFPL language in the next three chapters. In this chapter, we discuss most processing statements. We describe file handling statements in the next chapter, and in Chapter 8 we conclude our IFPL description with a discussion of the structure of IFPL programs and details on their interaction with the Idea Monitor.

The IFPL statements are of two types: definition statements for establishing registers, tables, and file parameters, and executable statements which accomplish the actual processing.

Definition Statements

Defining Storage Requirements

The REGISTER statement allocates memory storage. In it, you specify the amount of storage reserved, the data type, the position of the decimal point for numeric variables, and an optional initial value.

Defining Sub-Registers

In some cases, particular characters or groups of characters within a register have an independent meaning. For instance, in a time of day register, the hours, minutes and seconds are concatenated, but you may need to treat them separately. Similiarly, you may need to separate a check digit from an account number.

You can conveniently handle these requirements with the REDESIGNATE command. This command assigns a name to specified portions of a register. For instance, you might assign the name MONTH to the first two characters of the register DATE, the name DAY to the second two, and the name YEAR to the third two. The entire DATE register and each of its sub-registers, MONTH, DAY, and YEAR behave as independently addressable variables. You can move values from and into them, you can display them, and/or use them as elements of tables or in calculations.

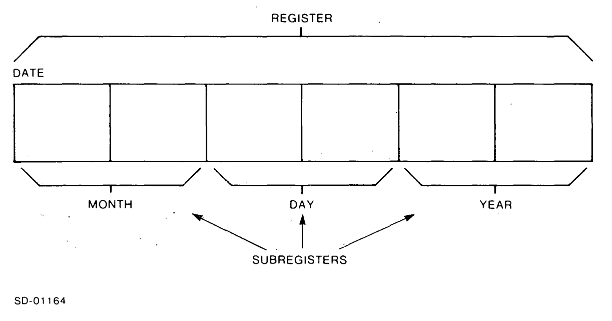


Figure 6-1. Subregisters

You can divide a register into any number of sub-registers, which may or may not overlap. It is not necessary to include all the bytes of a register when redesignating.

Defining Data Field Processing Modules

As more fully described in Chapter 8, a particular IFPL program module processes each data field. A PROCESS statement specifies the module, and, to save programming effort, reserves memory storage space for the data. This removes the need for a separate REGISTER statement to store screen data fields. The definition of the data field made during format preparation serves as the picture statement.

Creating Tables

You create tables with the TABLE statement by simply typing the table entries (literals or registers) in their proper order. You terminate the table with an ENDTABLE statement.

Executable Statements

Communicating with the CRT

The IFPL language has special statements for easy communication between your program and the terminal display. To take data entered at a console and store it in a specified location, use the STORE statement. To take data from a memory location and display it on the screen, use the DISPLAY statement. The MESSAGE statement displays error messages and instructions on the screen, and the LINK statement displays a new format.

Branches

IFPL also provides both conditional and unconditional branch statements. You write conditional branches as statement pairs consisting of a test instruction followed by an instruction specifying the branch condition and location.

The COMPARE statement compares two variables, then uses the result with an IF EQUAL, IF NOT-EQUAL, GREATER, or LESS statement to perform a branch.

The RANGE statement tests a variable against the range specified by two arguments. It is associated with the branch conditions In-range or Out-range.

For instance, an error checking routine might look something like this:

```
STORE MONTH
RANGE "1" MONTH "12"
OUT-RANGE ERROR
```

The variable MONTH, keyed in from the CRT, is checked against the range 1 through 12. If it is outside that range, the program branches to the routine ERROR. Otherwise, the system executes whatever statement follows OUTRANGE.

Though MONTH is checked against a fixed range in this example, the range also may be variable. In that case you would specify the registers in place of literals.

The LOOKUP statement, treated further under Tables, also may be followed by branching statements IF FOUND and IF NOT-FOUND. LOOKUP searches for a variable in a particular table. You might use it, for instance, to to check the validity of a password.

```
LOOKUP PASSWORD IN PASSTABLE
IF FOUND PROCEED
MESSAGE "INVALID PASSWORD"
```

In this example, if the PASSWORD (a reserved word into which the system automatically places the password used at sign-on) is contained in the table PASSTABLE (listing all valid passwords), processing will branch to the routine PROCEED. Otherwise a message is displayed and normal processing does not occur.

The ON-IOERR statement allows you to handle I/O errors with a minimum of extra instructions. You would use this statement to branch to the I/O error handling routine after any of the database access statements.

The GO TO statement causes the program to branch to a specified location unconditionally. The GO TO USING statement is the indexed version of the unconditional branch. It branches to one of a list of locations according to the value of the index. It allows you to write routines where your program dynamically determines the branching location. For an example, see TABLES, below.

The RETURN statement branches to the specified screen data field and its associated routine, via the Monitor. Its indirect version, RETURN USING, allows your program to dynamically specify the next data field to be processed.

Subroutines

The IFPL language also accommodates subroutines for common processing. The PERFORM statement branches to the subroutine named in its argument. You define the subroutine entry point with a SUBROUTINE statement and its exit with an ENDSUB statement. When the system encounters ENDSUB, processing will branch back to the instruction following the subroutine call. Subroutines may call other subroutines, and there is no limit to the number of nested levels. However, subroutines cannot call themselves nor subroutines through which they were called.

Tables

Tables are frequently the most efficient way to handle lists of data. IFPL supports table lookup routines with a number of special instructions. To do this, you predefine a table and all its elements in the definition (non-executable) portion of your IFPL program. Table elements are addressed as TABLENAME pointer where pointer is the register containing the element's position in the table. You may use this table address as an argument to the MOVE and DISPLAY commands. Thus you can move variables to or from tables and display them directly from tables.

Table elements may consist of literals or registers. In the latter case, the elements may be dynamically changed during program execution, but you may address them by either their register names or by their table position.

You can define up to ten tables, and address up to 99 elements per table.

To look up a variable in a table, use the LOOKUP instruction. If the system finds the variable, it will store its table position in the register specified. You can then use the table position to index a variable in a second table. You can also use it as an index for branching or linking via the GO TO USING RETURN USING or LINK USING statements.

One common application of the LOOKUP-- GO TO USING construction is in handling I/O errors. There are 12 error states, identified by non-consecutive number codes. In our example below, we have defined six of these error codes as elements of the table ERRTABLE. The reserved word IOERR contains the error code. The routine to determine the error and the point to which the program must branch might look like this:

```
LOOKUP IN ERRTABLE (INDEX) IOERR
GO TO E10 E22 E23 E24 E94 E96 USING INDEX
```

The system will use IOERR's position in the table ERRTABLE to select one of the six branches, E10...E96, specified in our GO TO USING statement.

Computation

IFPL has a complete fixed point arithmetic capability. The ADD, SUBTRACT, MULTIPLY and DIVIDE instructions perform computation. The result's accuracy depends on the size of the receiving variable. You can use numeric variables up to eighteen digits long and you can put the decimal point anywhere within the variable.

Linking to Other Formats

An IFPL program can also link to other formats and their associated programs. The LINK USING statement links indirectly, via a program memory location. The statement makes applications systems using format tree structures particularly easy to write. In a menu program, for instance, the LINK USING system can link to the correct format directly from the operator's input.

You can pass variables from an active program to its successors. You pass the data via a special system file or buffer. The PASS statement inserts the variables into the file. An ACCEPT statement in a successor program retrieves the variables. The file structure allows data passing between any set of programs run on the same CRT, even if other programs are interspersed between them. Thus you can implement a particular function by two, three or more consecutive IFPL programs. Intermediate data can pass from program to program, including from the first program to the last.

For example, suppose we have a menu that asks an operator to pick one of a list of alternatives by keying its item number. The program uses the item number to link to the proper format and its associated program. In the sample routine below, the format names are stored in a table called FORMTBL. The system moves the chosen format, indicated by ITEM, to the temporary register FORMAT, and performs the link.

```
STORE ITEM
MOVE FORMTBL (ITEM) TO FORMAT
LINK USING FORMAT
RETURN
```

The RETURN statement returns control to the IDEA Monitor, which will display the new format and load its applications program.

This routine, together with a process statement and some register statements, is all that is necessary to process a menu. The four statements above are the only executable statements required.

Source Library Files

You can maintain a library of standard IFPL modules as a series of source files. This is particularly convenient for definitions of standard data index structures and records which recur in all application programs using them.

To do this, write all standard program modules once and store them as individual source files. You can then insert them into any IFPL program using them with a single COPY statement. The IFPL compiler will expand the COPY statement to the full contents of the referenced file.

Generating Reports

IFPL programs can also generate data for reports.

To reduce the overhead in your application programs, the report printouts use formats just like the CRT displays. You create the formats interactively, under IFMT, in the same way. That is, you can use IFMT to create headings, data identifications and other literal information, as well as the definition and page placement of the data fields. Thus, you need perform none of these functions in your applications program.

All you need to do in your IFPL program is to file the data into a special system file in the order required by the format. Later, a utility program will read the file and map the data into the format as it prints it out.

Storing Print Data in the System File

Conceptually, the system print file consists of a serial string of variable data. Each item contains one or more data records which match the requirements of the format for the printout. The item's beginning and end in the file are marked, and the print utility will use these marks later as it maps the data into the print format.

You use the following three IFPL commands to generate the print file:

- The INITIATE PRINTING command inserts the start-of-printout mark into the system print file.
- The PRINT command stores a data record in the print file. A record corresponds to each page area (between tabular areas) of the printout format or to each tabular line of the printout.
- The TERMINATE PRINTING command marks the end of the printout.

All three of these commands reference a particular printout format. Thus your program can accumulate data for several printouts concurrently. However, you would have to initial and terminate each printout separately.

You may also generate printed output via COBOL or RPGII programs, or a Dasher printing terminal.

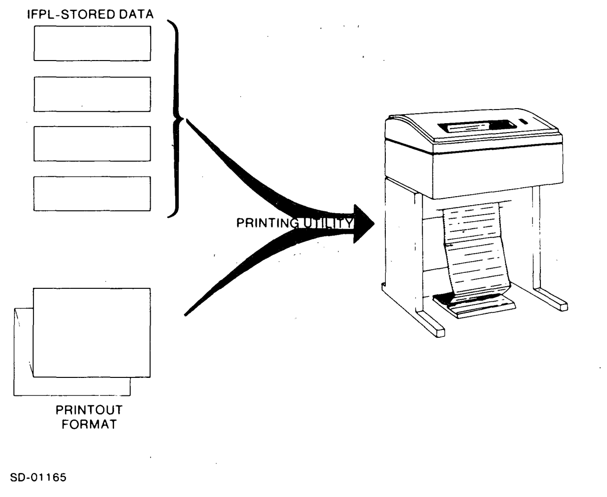


Figure 6-2. Printing Reports

Programming Keyboard Functions

As described in Chapter 2, the function keys allow you to select special operations, return to earlier portions of the format, and perform other functions. There are two types of functions; those which the system performs automatically, and those which you define in your applications program. The automatic functions are LOG OFF, END DATA, REPEAT PAGE, CHANGE MODE (used to exit from a scroll field), ERASE FIELD, DUP FIELD, AND BACKTAB. In addition, there are five function keys which you can define in your program.

Compiler-Directing Statements

Compiler-directing statements allow your program to sense external conditions and respond to them. Typical external conditions are:

- Striking a function key.
- Disconnection of a dial-up communications line.
- Attempted log-off.

The compiler-directing statements handle conditions such as these. All such statements have similiar syntax and functions. For example, to respond to a disconnected communications line, you would write the statement

```
ON DISCONNECT TAG
```

When a disconnect occurs, the program will automatically branch to the routine specified by tag. This routine will terminate the program cleanly.

All the function keys (except the editing keys CHANGE MODE, ERASE FIELD and DUP FIELD) can be sensed with similar statements.

Three of the function keys -- LOG OFF, END DATA and REPEAT PAGE -- do not need to be implemented in your program. They are functions of the Monitor, as described in Chapter 2. Therefore, your program does not have to sense or implement them. However, there are cases in which your program must control the use of these keys. For example, your program might disable the LOG OFF key during a portion of its processing; e.g., to prevent an operator from exiting from a program before updating all of a number of related records.

In other words, let us assume that your program updates the records in the routines for fields four, five and six. If someone tries to exit from the program after entering field four and before completing field six, it will result in incompatible records. Thus you must disable the LOG OFF key during the processing of those fields. Your routine might look like this:

```
REGISTER FIELD 9(2)
ON LOGOFF TEST-FIELD

TEST-FIELD:
RANGE "4" FIELD "6"
IF IN-RANGE LOGOFF-ERROR
QUIT

LOGOFF-ERROR:
MESSAGE LOG OFF NOT ALLOWED IN THIS FIELD
RETURN USING FIELD
```

In this routine, our program branches to TEST-FIELD every time the LOG OFF key is struck. The current field (automatically maintained by the reserved word FIELD) is range checked to determine if a log off is allowed. If it is not, we display an error message and the program returns to the current field. If a log off is allowed, the program executes a QUIT.

Inactivity Clause

To better utilize system resources, your program can limit the amount of time an operator can spend entering data into a single field.

That is, the statement INACTIVITY CONSTANT IS minutes defines the outside limit for a reasonable waiting time. Each program can set it differently, to anything from one minute to two hours, depending on the particular task.

When the inactivity constant is exceeded, the program will branch to the routine defined in the ON NO-ACTIVITY tag at which point it will do whatever is appropriate. For instance, it may log the terminal off.

NOTE: Operator commands can only occur at data fields. At all other times, keyboard entry is disabled. Thus you need take no special precautions against exit from the program if each program module completes all housekeeping tasks.

End of Chapter

Chapter 7
IFPL File Handling Statements

The database structure and access capabilities of the INFOS system are available to your application programs through the IFPL definition and operation statements. The definition statements specify the file structure to your IFPL program, and the operation statements store, retrieve, update, and delete data. Note, however, that you cannot create files via an IFPL program. Instead, you must create them beforehand with the ICREATE utility.

Database Definition

To build an INFOS structure, you must define the files that you want to access, the index paths, key lengths, and record parameters you want to use.

Files

You must use a file statement to identify the files your program will use. Any IFPL program may open up to three files, using a maximum of fifteen index and subindex levels. You can apportion the fifteen levels among the three files in any way. Note here that the IFPL language uses the word *file* synonymously with *index*. Thus, the file name is the name of the highest level index.

This restriction on the number of files and index levels does not include system files. Thus, in addition to any files your program opens explicity, your program can also use the system-maintained Transaction and Common files.

Indexing Path

You must specify the indexing path required to reach each record type by including a statement for each subindex level. Thus, you could specify a three-level salesrep file indexed by region, territory, and salesrep's name by these two statements:

```
SUBINDEX FOR REGION IS TERRITORY
SUBINDEX FOR TERRITORY IS NAME
```

These two statements establish the index chain-- REGION is linked to TERRITORY, and TERRITORY is linked to NAME.

Key Length and Duplicate Keys

For each index or subindex level, you must write a KEY statement which specifies the maximum length of the keys contained in the index. The INFOS system uses this definition to create subindexes as they are required.

The system also allows you to use duplicate keys at any level. If you want to allow duplicates, you must use the DUPLICATES COUNTED statement to define a register for storing the duplicate occurrence count.

Record Description

Next, you must associate a record type with its index or subindex. For example, your file might contain a record describing monthly sales; the statement

RECORD FOR NAME IS SALES

would establish this.

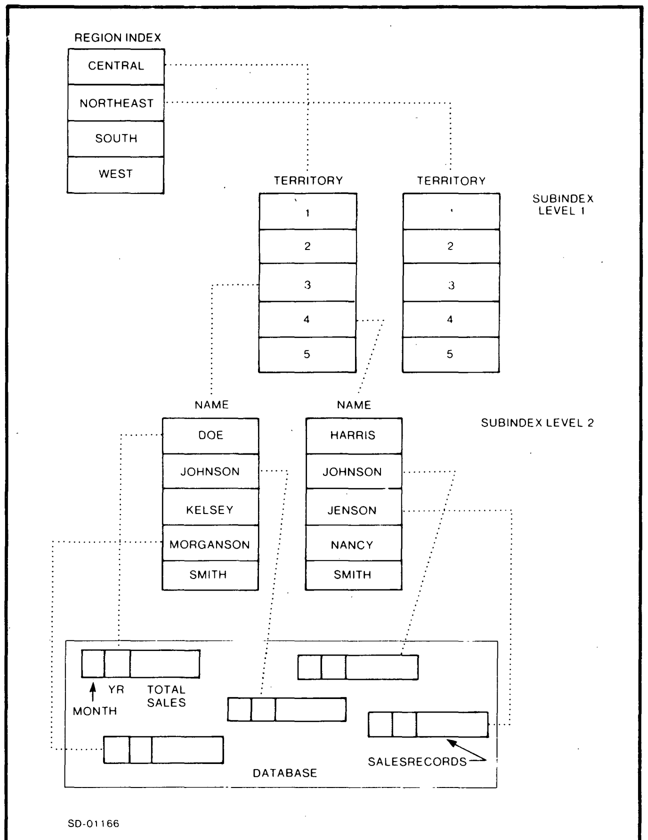


Figure 7-1. Salesrep File

Each record type has a fixed format and contains a specific number of variables arranged in a predefined sequence. (These variables are also known as fields; we have used the word variables to avoid confusion with screen data fields.) Once you have accessed a file, you can manipulate the variables within it as you can program variables. It makes no difference whether you declare a variable in a register, calculate or input it from a CRT, or read it from a database record. The variables are packed and are not separated by delimiters. Thus you will require a template to separate the record into its constituent variables. Continuing our example, the sales record contains three variables--month, year, and total sales for the month. A typical record might look like this:

11760046057

You would specify length and constituent parts in the following statements:

```
LENGTH IS 11
INCLUDES MONTH 1 2 ASCII
INCLUDES YEAR 3 2 ASCII
INCLUDES SALES 5 7 ASCII
STOP
```

The third INCLUDES statement specifies that the variable SALES starts at the fifth character, and is seven ASCII characters long. From the other INCLUDES statements we can determine that, for this record, the month is 11, the year is 76 and sales are 46,057.

The INCLUDES clauses do not reserve space; they only define record fields. You must match each INCLUDES field that you want to use with a corresponding program variable.

You can associate any number of different record types with a particular index structure. For instance, you could also have name and address records associated with each salesman's name; other records might be associated with the region and territory. You would access the latter two through one and two indexing levels, respectively.

To use these additional records, you need not re-define the index path statements, but only the additional records.

Managing Subindexes

For programmers with a good knowledge of the INFOS data manager, IFPL provides two facilities for managing subindexes. These facilities are optional; they improve storage efficiency but do not otherwise affect functionality.

Define Subindex

This statement allows you to specify subindex parameters such as node size and partial length. The IFPL statement is PARAMETERS FOR subindex.

Link Subindex

Keys within an index may be logically synonymous. For instance, consider the file diagrammed in Figure 7-1. Suppose that the "Northeast" region is also known as "Atlantic." To access all records presently accessed by the key "Northeast" via the key "Atlantic," you could add the key "Atlantic" to the region index and link it to all Territory indexes presently linked to "Northeast". In that way, Northeast and Atlantic share all the subindexes. No extra space is required since they are not stored twice.

The link facility is further described under the ESTABLISH LINK statement in the *Idea Programmer's Reference Manual (AOS)* (93-000151)

Data Retrieval

You use a FIND statement to retrieve variables stored in the database. This statement reads the record from storage into processor memory. Arguments to the statement define the record type and specify the key or keys needed to access a unique record.

After the system executes the FIND statement you can manipulate the variables within the record like any others. The INCLUDES definition is all that you need to define their names and positions within the record. You can move the variables to other memory locations, display them on the terminal, and use them in calculations.

You can implement all the access modes described in Chapter 5 with versions of the FIND statement, as shown in Table 7-1.

Retrieve Key

In Idea you do not need to include the key and the duplicates count within the record because the system stores them within the database.

When you access a record sequentially (with FIND NEXT or FIND PREVIOUS), with an approximate key (FIND NEAREST), or with a generic key (FIND BEGINNING), you may not know the actual key and its duplicates count. However, you may use this information for a number of reasons; for instance, to delete a record. You can return the record's lowest-level key and duplicates count with the RETRIEVE KEY statement.

Verify

In some circumstances, you will not need the record itself, but you will only need to determine whether it exists. For instance, an error checking routine might check an input account number against those in the file to verify that it exists or, perhaps, to insure that it is not already assigned. IFPL'S VERIFY command will perform this function by checking the database for the existence or non-existence of a particular record. It requires fewer disk accesses than the FIND command and thus improves overall response time.

Table 7-1. File Retrieval Statements

Access Mode	IFPL Statement	Definition
Full Keyed Access	FIND	Retrieves record exactly as specified.
Generic Key	FIND BEGINNING	Retrieves record for which you give only beginning of key.[1]
Approximate Key	FIND NEAREST	Retrieves record whose key equals or follows the specified key.
Sequential Access	FIND NEXT FIND PREVIOUS	Retrieves the record following (FIND NEXT) or preceding (FIND PREVIOUS) the last record accessed. In effect, this converts the file into a sorted serial file.

Note 1. In FIND BEGINNING statements with more than one level of keys, only the lowest key can be partial. Similiarly, in multiple-keyed FIND NEAREST statements, only the lowest key is approximate.

Record Locking

If you must protect an accessed record from change or access by other programs operating concurrently, you may lock the record. Each of the FIND commands described above has a variant which prohibits access to the record by other users. For instance, the locking version of the FIND command is FIND AND HOLD. The other commands use parallel syntax.

The system will automatically unlock the record when you refile or delete it (see below). Alternately, you can explicitly unlock the record with a RELEASE statement.

Data Storage

You store variables in the database by reversing the operations required to access them. That is, you first assemble the record in processor memory by specifying the values of all record variables that you want to change. Then you store the record in the database.

There are two storage instructions: FILE-NEW and REFILE. FILE-NEW creates a new record; REFILE updates an existing record with the new values of the variables. You specify arguments with both instructions which supply the record type and the keys by which you want it filed.

Inversion

If you want to access a record by more that one index path, you must invert it through alternate paths after you file it.

For example, consider the cross-indexed structure shown in Figure 7-2. In this structure, you can access each record via either the index path A,B,C, or through D,E. When you store a new record via one index path (for instance A,B,C), you must "invert" it through the other path (D,E). Inversion means that you must update the keys in the second path so that you can later access the record by that path. You do this with the IFPL command INVERT, followed by record and key arguments.

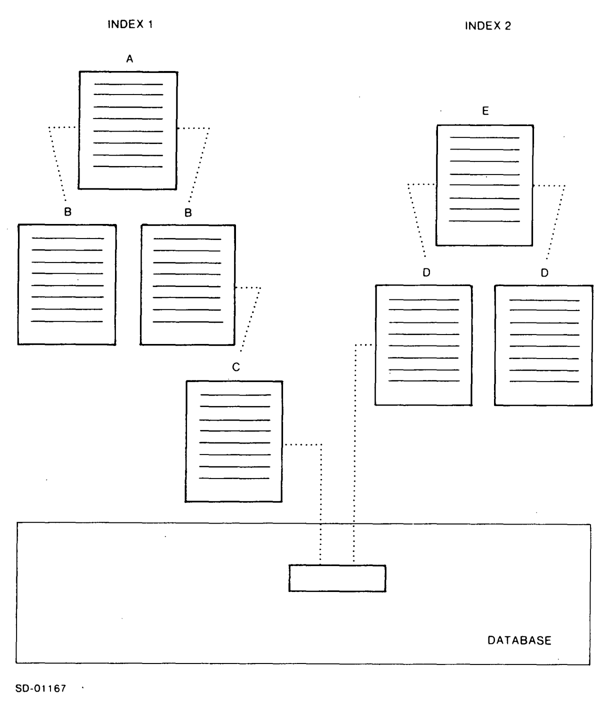


Figure 7-2. Inversion

Deleting Records

To delete records from the database, you can use either the DESTROY or REMOVE commands.

The DESTROY command physically deletes the record from the database and frees the storage space it occupied for reuse. After the system executes this command, the record is no longer accessible.

The REMOVE command performs a logical delete. That is, it flags the record as deleted but leaves the record accessible. When you access a logically deleted record, the system sets the reserved word IOERR to 96 to indicate the record has been logically deleted.

Although the REMOVE command does not actually perform a deletion and free disk space, you might use it in preference to the DESTROY command for two reasons. First, it is safer than the DESTROY command; its action is reversible. Second, it is also faster than the DESTROY command. Thus, most of your IFPL programs might use the REMOVE command, then periodically a separate program might physically delete all logically deleted records.

The REINSTATE command will restore a record that has been logically deleted.

Like the other record reference commands, you follow these commands with record type and key arguments.

End of Chapter

Chapter 8
Structure of IFPL Programs

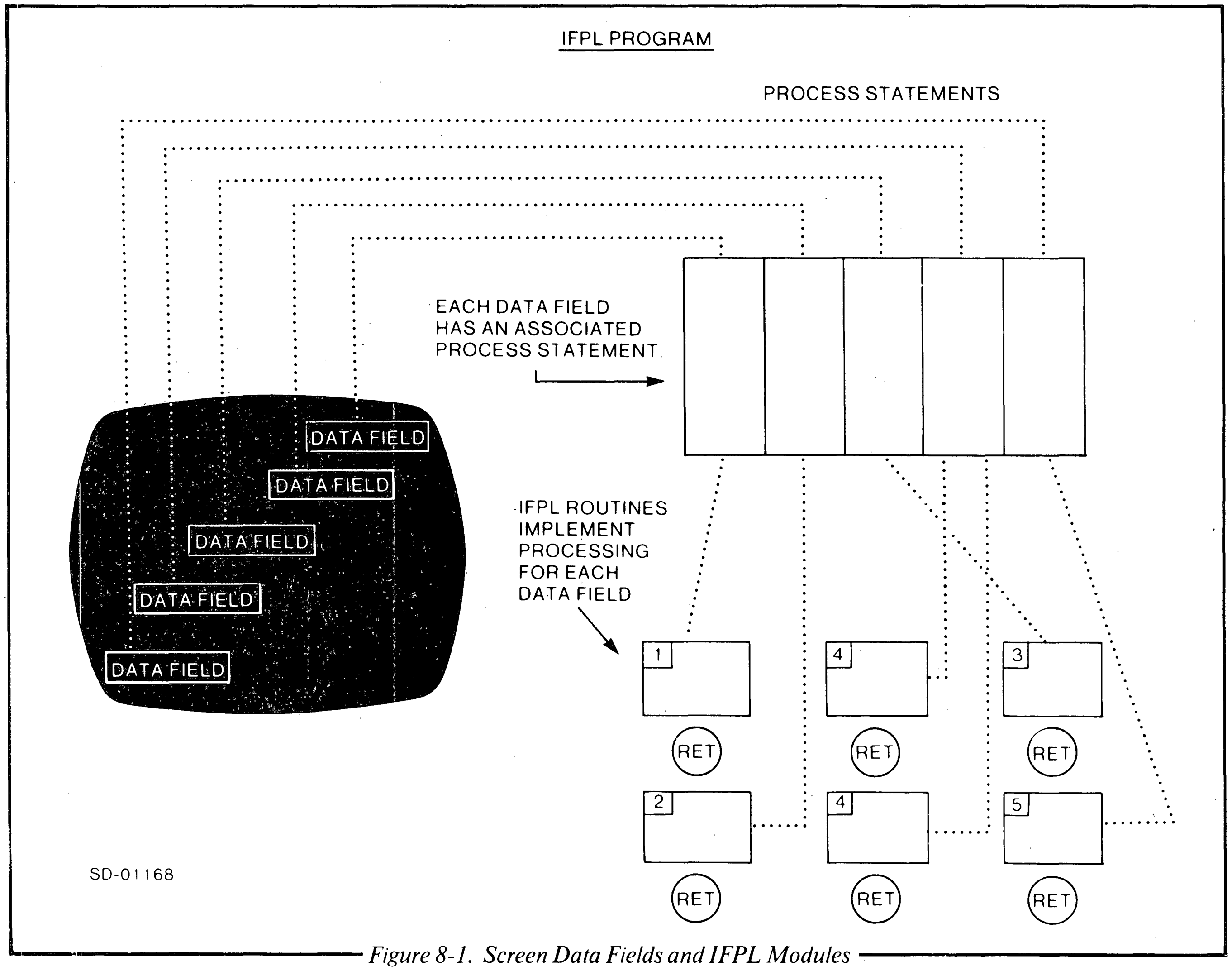


Figure 8-1. Screen Data Fields and IFPL Modules

Each IFPL program you write will support a specific screen format. Much of the ease and speed with which you can write IFPL programs depends on the close correspondence between IFPL structures and their analagous format structures. The program relates to the format in three principle ways: through Field Orientation, Monitor Interaction and processing flow shown in Figure 8-1.

Field Orientation

IFPL programs are field-oriented--that is, each screen format data field may have its own unique processing module within the IFPL program. (A module contains the procedure for handling a particular field's data.) Fields with both display and edit attributes can have two modules--one for processing display information and a second for processing edit information. Fields with similar processing requirements may share a single module.

Note, however, that if a data field has no Edit or Display attribute, it will have no corresponding module in the format's IFPL program. From the standpoint of the program, processing will occur as if the field did not exist. However even though you may not include a data field in your program, the Monitor will recognize it, position the cursor to the field, and check the input for errors. The Monitor will also file the data in the Transaction file if you specify the OUTPUT attribute for the field.

To assign the processing module (or modules, for display and edit fields) to a data field, use a PROCESS statement.

Monitor Interaction

The system passes control from the Monitor to the next IFPL program module and back to the Monitor on a field-by-field basis. The Monitor thus directly handles the overhead associated with terminal communications. It requires no support from your program.

The Monitor thus directly handles the overhead associated with terminal communications. It requires no support from your program.

Here is how the system processes a single data field.

First, the Monitor moves the cursor to the data field. If the field is an Edit field, the Monitor will wait for the operator to enter the required data; it will then check the data for errors. After the error checks are satisfied, the Monitor will store the entered data in a reserved area. From there, it is available to your IFPL program via the STORE command. To pass control from the Monitor back to the IFPL program module specified for that data field, you use a PROCESS statement.

If the field is a display field, the Monitor will pass control immediately to the specified IFPL program module. Actual display of data in the field will not take place until the system has executed the IFPL module.

Processing will then proceed under IFPL control. Processing may include database operations, arithmetic operations, execution of subroutines, branches to other portions of the IFPL program or any of the other statements described above. Note that the system will interpret any STORE or DISPLAY instructions as pertaining to the current field, regardless of where they occur within your IFPL program. This is true even if the program branches to code within a module associated with another field.

Control will pass from the IFPL program back to the Monitor when the system encounters a RETURN statement. If the field is a display field, the Monitor will display any information specified by your IFPL program. The Monitor will then position the cursor at the next logical data field and repeat the process.

Processing Flow

Each time the Monitor gains control, it will position the cursor at the next logical field and re-enter the IFPL program at the module specified for that field. Thus, processing flow is field-determined. However, there are two distinct aspects to determining processing flow; the system must determine which is the next logical field filled and then which module is effective (fields with both display and edit attributes are associated with two modules).

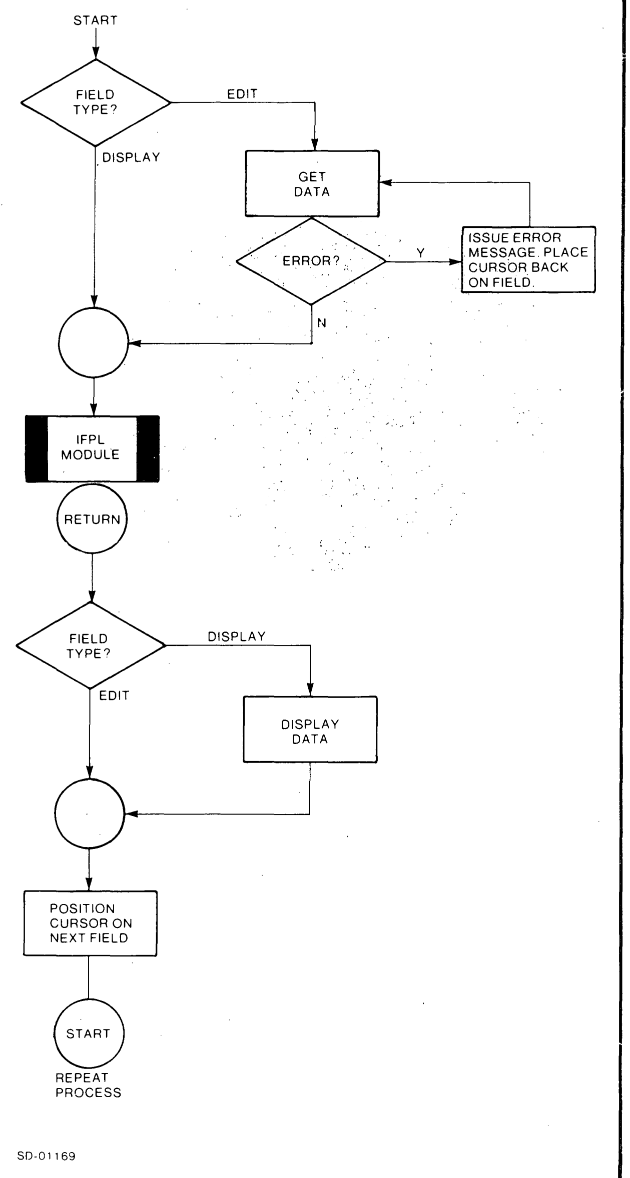
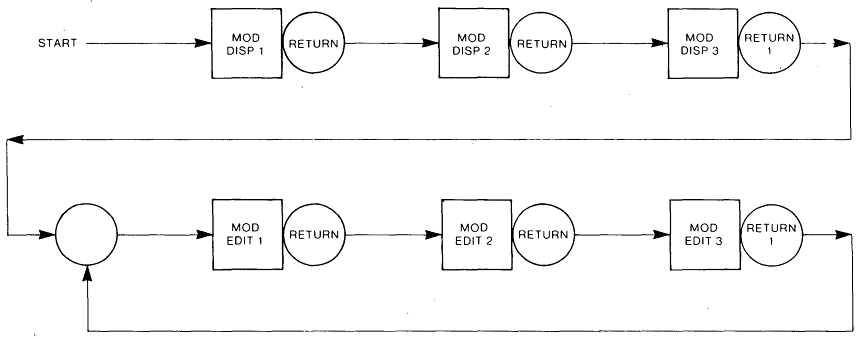


Figure 8-2. Processing a Single Data Field



Display and Edit Processing Order

Processing for this three-field format starts at the display modules. Because DISP3 returns to field one, the fields are processed again, this time by their edit modules. EDIT3 also returns to field one and the fields are again processed by their edit modules.

SD-01170

Figure 8-3. Display and Edit Processing Order

The Next Logical Field

Generally, the next logical field is the next physical field on the screen, following a sequence of left to right and top to bottom. The logical sequence differs from the physical sequence in the following two cases, given in the order of priority.

1. You may have specified the next logical field in a RETURN statement, in which case the Monitor will position the cursor to the specified field and branch to its effective IFPL module.

   You should use this form of the RETURN statement to exit from loops under program control. It is the best way to return to data fields whose display processing has been completed but whose edit processing is pending.

2. The logical field following the last data field in scroll area is the first field of that scroll area. Thus, a scroll area creates an implicit loop in the program. You must explicitly exit from the loop in your IFPL program or via an operator command if you want to process data fields outside the scroll area after processing within the scroll area has started.

Display and Edit Processing

After the Monitor determines the next logical field, it will enter your program at the effective module specified in that field's PROCESS statement. If you specify only one module, the system will process that module every time it encounters the field. If you specify two modules, one for display and one for edit processing, the system will determine the effective module according to the following rules:

- The first time the system encounters the field, processing takes place at the display module.
- The second and every subsequent time the system encounters the field, processing takes place at the edit module.

Note, however, that nothing inherent in the system assures that a field will be encountered a second time. Your program must explicitly loop back by using a RETURN statement.

Note also that you can reset a display and edit field to DISPLAY by executing a RESET statement.

If a scroll field has both the 'Display' and 'Edit' attributes, the system will process it as shown in Figure 8-3.

The first time the system encounters the field, the display module will process it. If the field is encountered again, on the same line, the edit mode will process it. Once the system enters the next scroll line, it treats the field as if it were encountering it for the first time. That is, the display module processes the field first, then (if encountered again) the edit module.

Starting Address

Implicit in the processing order described above is the starting location of your IFPL program. The program will start at the effective module of the first screen field. Occasionally, however, you may want to perform some initialization or other processing before the first field. In that case, you should create a single character display-only field at the beginning of the screen, then use its associated module for initialization or whatever.

At the End

If the module for the last field ends with just a RETURN (without a field number), the Monitor will link to a new format, as specified during preparation of the current format. (Specifically, the format you entered in answer to the question LINK? asked at the end of IFMT.) You can use this facility to link to the next task or to the system's ground state.

If you specified no format in response to the LINK? question, control returns to the Monitor. It will flush the screen and display either the ground state format or the question FORMAT? to the operator.

Other Considerations

Because field order in the screen format usually determines the processing flow, arrangement of data fields is quite important. If you arrange data fields in the order in which information is available -- as, for instance, the blanks on a 1040 income tax form -- the supporting IFPL program is easy to write. Conversely, if an IFPL program is difficult to write, it may help to rearrange the data fields in a more logical order.

The GO TO and the RETURN statement are sometimes confusing because both statements cause processing to branch to another portion of your IFPL program. GO TO will branch to the location specified by its argument while RETURN will branch to the location specified in the next logical PROCESS statement.

However, there is a crucial difference between the two. GO TO will branch immediately, and no matter where it branches, the current screen field will remain the same. Furthermore, the system will interpret all DISPLAY and STORE commands with respect to the same field.

The RETURN statement, on the other hand, will branch via the Monitor. When your IFPL program regains control, the effective data field will be the next logical field. The Monitor interprets all DISPLAY and STORE statements with respect to a new data field.

End of Chapter

Chapter 9
Summary

Idea is an integrated software package for the development and operation of interactive data entry/access systems on Data General's ECLIPSE-line systems.

The major operating and development features of Idea are summarized below.

Operation

Operation is screen-format driven and interactive, using an on-line database. Many operating features make the system particularly easy to learn and to operate, including error detection, prompting, and dedicated keyboard function keys. In addition, your applications software can easily incorporate menus and other interactive operator aids. As a result, you can move data entry and retrieval functions to operating departments with significant improvements in responsiveness, reduced error rates and cost savings.

Formats

You express system data transactions with as many screen formats as you need. These formats can link to other formats automatically, either unconditionally or as a result of operator action.

You prepare formats interactively under the Idea utility, IFMT, by typing the format literals and data fields as you want them to appear to the operator.

You can specify data fields as Alphabetic, Alphanumeric or Numeric. Numeric fields may have a floating currency symbol, an arithmetic sign, a decimal point leading zeros suppressed, and a check-protect character. You can assign data fields one or more of the following attributes: Display, Edit, Output, Required, Full, Secure, Auto-Dup and Auto-Entry.

INFOS Database Manager

Idea uses INFOS DBAM database structures, which means that the system stores data in records which it retrieves via index and keys. These indexes may be multi-level, and you may cross-index data records. In addition to full keys, you may use approximate keys and generic keys. The system also supports both forward and backwards sequential access.

The INFOS system is also the key to compatibility between Idea programs and other systems running on the same processor. Because Idea uses the standard INFOS system, all data is equally accessible to programs written in COBOL or other Data General languages through the same data structures and access keys.

Applications Programs

You write applications programs in IFPL, a language specifically created to make format-related programs easy to write. IFPL programs consist of modules tied to specific screen data fields. This structure allows the runtime Monitor to perform many of the overhead functions required for interactive screen I/O. Table 9-1 lists all the IFPL statements by category.

Table 9-1. IFPL Statements

Definition REGISTER REDESIGNATE PROCESS TABLE ENDTABLE	**Unconditional Branches** GO TO GO TO USING RETURN RETURN USING	**File Storage** FILE-NEW REFILE REMOVE DESTROY REINSTATE INVERT ESTABLISH LINK
Communicating With a CRT STORE DISPLAY MESSAGE LINK	**Linking** LINK USING PASS ACCEPT	**Record Locking** HOLD RELEASE
Subroutines PERFORM SUBROUTINE ENDSUB	**Computation** ADD SUBTRACT MULTIPLY DIVIDE	**Hard Copy** INITIATE PRINTING PRINT TERMINATE PRINTING
Conditional Branches COMPARE IF EQUAL IF NOT-EQUAL IF GREATER IF LESS RANGE OUT-RANGE IN-RANGE LOOKUP IF FOUND IF NOT-FOUND ON-IOERR	**File Definition** FILE SUBINDEX KEY RECORD LENGTH INCLUDES REDEFINES PARAMETERS FOR SUBINDEX NODE SIZE PARTIAL LENGTH DEFINE SUBINDEX DUPLICATES COPY	**Mediating Keyboard Commands** ON BACKTAB ON END OF DATA ON ESCAPE ON LOGOFF ON FUNCTION ON REPEAT ON SCREEN
	File Retrieval FIND USING FIND NEXT FIND PREVIOUS VERIFY RETRIEVE KEY RETRIEVE HIGH KEY	**Other Statements** MOVE LEFT RIGHT COPY QUIT RESTART RESET INACTIVITY CONSTANT ON NO-ACTIVITY LOG ON DISCONNECT ON LINE-ERR PRIORITY

FILES also supported
RESET statement
USING uid
with ... ON-OVERFLOW ident)
FIND BEGINNING ~~WITH~~
NEAREST
REFILE

End of Chapter

Index

Within this index, "f" or "ff" after a page number means "and the following page" (or "pages"). In addition, primary page references for each topic are listed first. Commands, calls, and acronyms are in uppercase letters (e.g., CREATE); all others are lowercase.

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 ______________

SD-00742

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

SD-00742A STAPLE