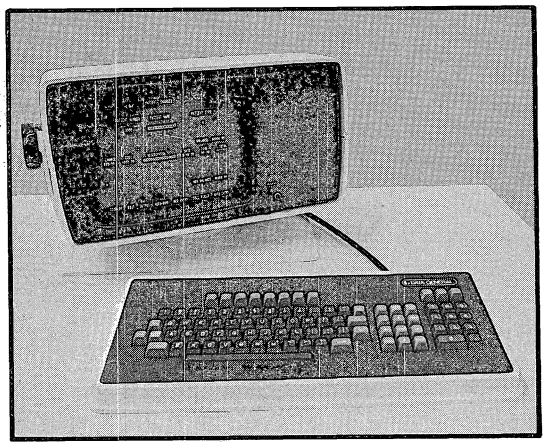
DataGeneral



Idea

Interactive Data Entry/Access

Concepts and Facilities

Idea

Interactive Data Entry/Access

Concepts and Facilities

045-001-00

Data General Corporation, Southboro, Massachusetts 01772

On The Cover

The CRT is Data General's model 6053. Its screen swivels and tilts for operator comfort. The keyboard is free standing and can be positioned wherever convenient.

The display shows an Idea data entry format in preparation. Formats are prepared directly on the CRT, as described in Chapter 3.

Printed in U. S. A.

TABLE OF CONTENTS

PREFACE

The Concepts and Facilities manual outlines the general capabilities and underlying principles of the Interactive Data Entry/Access (Idea) system. It is meant for readers with several different interests.

Audience

This manual is written both to the general reader who wishes to form an overall impression of Idea and to the technical reader who will be implementing Idea systems. The first five chapters of the manual describe the capabilities of Idea and will be of interest to all readers. Chapters six through eight describe the Idea language in some detail. They are primarily intended for readers with some programming background.

Organization of Manual

An overview of the entire system is presented in the Introduction. Runtime operation and examples of formats as the operator sees them are presented in Chapter two. Chapter three describes format structures and the interactive utility under which formats are prepared. In Chapter four, an example of an applications system shows how formats are integrated to form larger systems. An example of how a program supports a format is shown. The versatility and power of INFOS database structures is discussed in chapter five.

Chapters six, seven and eight contain a discussion of Idea programming language. All major Idea statements are discussed. The special modular, format-oriented nature of the language is covered in chapter eight.

Idea Documentation

The Concepts and Facilities manual is only part of the entire Idea documentation package. All Idea documents and relevant INFOS documents are listed below.

Title	Order No.
Idea Product Brief	012-301
Idea Systems Brief	012-302
Idea Concepts and Facilities Manual	045-001
Idea Reference Manual	045-002
Idea Release Notice	085-047
Introduction to INFOS	093-113
The INFOS Storybook	093-199
INFOS Utilities User's Manual	093-182
INFOS System Planning Manual	093-115

1. INTRODUCTION

What is Idea?

Idea the Interactive Data Entry/Access (Idea) system, is an integrated software package for developing and operating data systems. Idea applications systems are transaction-driven, are on-line and operate interactively on CRT or hard-copy terminals. The development of these applications systems has been made particularly simple. Formats are generated interactively, without programming. Applications programs are written in a Cobol-like language specially designed for the interactive, transaction-driven environment of Idea.

Idea runs on Data General's commercial Eclipse systems under RDOS, the Real Time Disk Operating System. Idea utilizes the powerful INFOS database manager. Thus Idea is fully compatible in both operation and data structures with other Data General commercial Eclipse software.

Idea consists of three major elements--a format generator, an Idea language and an Idea monitor. Each is discussed in greater detail below.

Idea Format Generator—IFMT. Screen formats are generated interactively and require no programming. Under IFMT, formats are prepared directly on the CRT screen. The format is typed as it will appear to the operator. Parameters are specified through a question and answer dialogue. IFMT allows formats to be prepared by non-specialists, so that formats may be designed by people acquainted with the application rather than with programming.

The maximum number of formats a system may have is very large. Ultimately, it is limited only by the available database capacity.

Idea Field Processing Language—IFPL. Application programs are written in Idea's Field Processing Language, IFPL, a Cobol-like language optimized for data transaction programs. IFPL includes statements for branching, arithmetic operations, special file handling instructions and statements for direct interaction with CRT screen formats.

To reduce programming effort and time, many functions which are traditionally handled in the applications program are handled automatically in Idea. For instance:

- The IFPL program does not include the format. The format is displayed directly from the file created by the format generator.
- The applications program does not have to move the cursor or accept data from the CRT on a character-by-character basis. That's done by the monitor. A single instruction in IFPL suffices to store data input on the CRT. Similiarly, the program can display data with a single statement. It can display a message with a single statement.
- Checking CRT-entered data for illegal characters is performed by the monitor. Resulting error messages are also handled by the monitor. Only validated data is passed to the applications program.

Modularity. Idea is inherently modular. Idea's fundamental organizational unit is the format. Each format is supported by a separate IFPL program. The program itself consists primarily of processing procedures—each procedure related to the needs of a particular format data field.

This modularity makes applications easier to plan and develop. Later, when changes to the system are required, they tend to be isolated to specific programs and formats. Similiarly, extensions to the system can be made by adding programs and formats.

Operation. Idea applications programs operate under the Idea monitor, IMON. Up to 16 terminals can operate concurrently running any combination of the same or different formats and programs.

The monitor is transaction-driven. The terminal operator can choose and run any one of the available formats totally independently of what may be running on other terminals.

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

The interactive, transaction-driven nature of Idea applications systems allows considerable latitude in the way data entry and retrieval is organized. In many cases, data capture may be decentralized and moved to its source, with attendant improvements in timeliness, reduction in errors, and decreases in cost. Similiarly, data inquiry can take place directly in operating departments. Long, bulky, expensive and out-of-date paper reports can be largely eliminated.

INFOS, the Database Manager.

Data is entered and accessed from the database under the control of INFOS. INFOS uses a DBAM (DataBase Access Method) file structure that is particlarly well suited to the multi-user, multi-program sharing of the same data files.

Data records are key-accessed through a versatile indexing structure. Indexes can be multi-level; records can be cross indexed through a number of indexing paths.

The database is always accessible and up-to-date. Entered data is posted immediately. Accessed data is always the latest available. Delays associated with off-line input, batch updating of files, printing and distributing of reports are eliminated.

Concurrent Processing

The Idea runtime system occupies only one memory ground. Thus it can operate concurrently with other tasks.

The second ground can be used for batch data operations. Programs written in Cobol, RPG or other languages may be run. Via INFOS, they can access the same database and use the identical indexing structure.

Programs in this ground can interface the Eclipse system to a larger host DP installation or to other Eclipse and Nova systems. A wide range of communications hardware and software packages are available to implement this kind of inter-system communication.

The second ground can also be used for program development. The ground may be used for compiling IFPL programs as well as programs in other languages, such as Cobol, RPG or Fortran. It may be used to generate or edit Idea formats. Printing of reports of data generated by the runtime system can also be done in the second ground.

The System Hardware

Idea runs on commercial Eclipse systems comprised of the following major components.

Operator Terminals. Idea supports up to 16 terminals. These can be any mix of CRT's or hardcopy terminals, as required by the application.

Terminals can be used remotely, via dial-up telephone lines. In the latter case any number of terminals adequately served by sixteen lines can be supported.

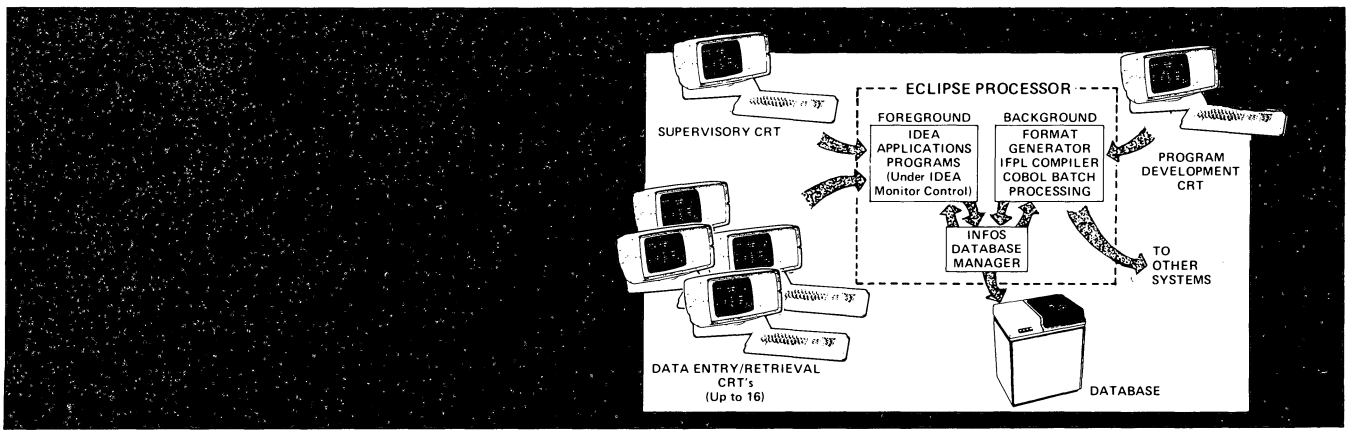
Supervisory CRT. The Idea system supervisory console is used to start and stop the system and monitor CRT activity.

System CRT. On systems with a second ground, an additional CRT is used. It controls background processing running under the real-time disk operating system, RDOS.

The Commercial ECLIPSE Processor. The commercial Eclipse 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.

Database Storage Devices. A variety of disk storage devices can be used --ranging in capacity from 2.5 to 90 megabytes. The database can span up to eight physical devices of the same or different types for a total capacity of up to 720 megabytes of on-line storage.

Magnetic tape can be used for archival storage and for inter-system data interchange. Industry-standard tape formats are supported.



2. OPERATION

Idea can support up to 16 data entry/access terminals operating concurrently. The terminals can run any combination of the same or different formats and their associated applications programs. A format and its program is called up on the system with the sign-on function.

Accessing Formats—The Sign-On Function

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

SIGN-ON DISPLAY

A format can be called from any unused terminal in the system. The operator strikes *Sign-on* on the terminal. The terminal responds with the sign-on display. The operator types an identification code or a password and the name of the format to be used. The system responds by displaying the format and positioning the cursor on the first field the operator must fill out. (Note 1)

Data Entry

The operator enters data from the keyboard. Keyboard errors can be corrected with the *Rubout* key. The cursor can be positioned anywhere within the field. Characters can be overstruck. The type and number of characters required by the field are displayed for the operator's reference by a prompt in the lower right corner of the screen. When entry for the field is completed, the operator strikes the *Enter Data* key.

The Idea monitor, IMON, examines the data. If it discovers an error, it displays a message and repositions the cursor to the field. If the data meets all its defined criteria, the monitor makes the data available to the applications program. (Applications program processing is discussed in detail in later chapters.) If the data field has the attribute *Output*, the data is also output to the transaction file, a file created and maintained by the monitor. The monitor positions the cursor to the next field and the process repeats.

When data entry for the format is completed, the format is flushed from the screen. The next logical format, if one were defined, is displayed. Or, the operator is asked to choose another format.

Operator Commands

The usual sequence of operations described above can be modified by several keyboard commands. Each is effected with a keystroke.

Backtab moves the cursor to back to the previous data field. It is used to re-edit a field.

End of Data ends the present format and displays the next format, if one has been defined.

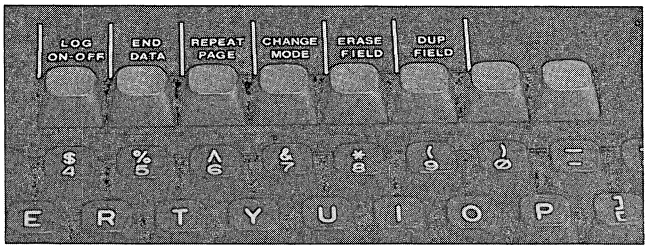
Log Off ends use of the terminal. The terminal is now available to another user.

End Scroll Area terminates processing of scroll fields (see below) and moves the cursor to the first field beyond the scroll area.

Escape terminates processing of the input operation of the present field. Processing control is given the applications program. Generally, the program's escape processing will perform any required housekeeping tasks and terminate the present format. However the escape function can be defined in other ways also, as determined by the program designer.

Erase Field is used for corrections. It deletes the entry and positions the cursor on the field's first character position.

Dup Field is used in scroll mode. It duplicates the data in the field above.



OPERATOR KEYBOARD (6053 TERMINAL)
The top row of keys is dedicated to command functions. The operator has no special function codes to remember.

NOTE 1

Password handling must be implemented in the applications program. The ID used in calling a format is available to the applications program in the reserved word *Password*. The terminal number is available in the reserved word *CRT*. The applications program can use these 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, for instance the menu, as the *ground state* format. If one is designated, the terminal operator no longer can choose a format directly at sign-on or at any other time. Instead, the ground state format is displayed. It links to other formats in the system. The links may be made conditional on password and CRT number.

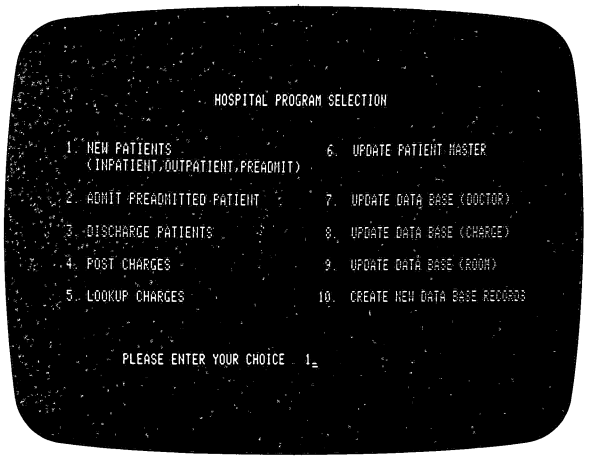
Examples of Formats

Shown below are examples of the kind of format structures and operations that can be created 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, the menu can be used as the system 'ground state' format—providing access to all other formats in the system. Then, at sign-on, the menu format is the only one the operator need remember. On the other hand, experienced operators can bypass the menu and access the required format directly.

The menu format is implemented by an applications program. It is extremely simple and short. It requires just half a dozen IFPL program statements!

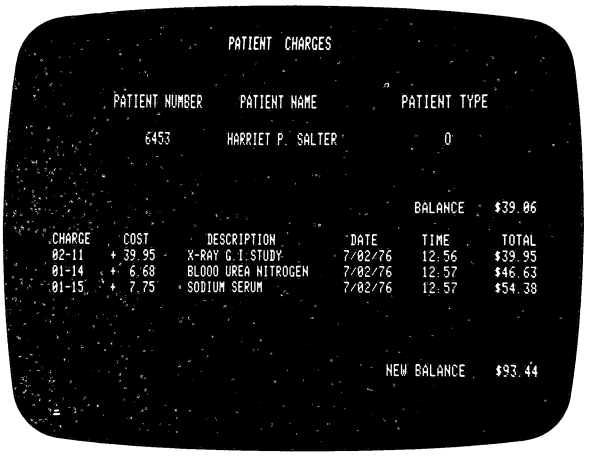


MENU FORMAT

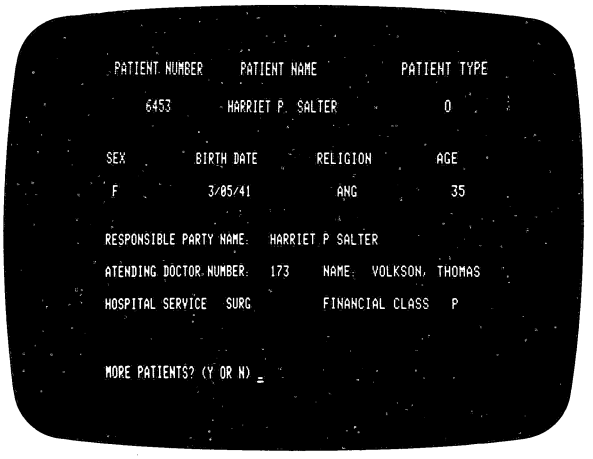
Scroll Fields. Scroll fields are convenient for entering tabular material. For instance, the patient charges list in the format is a scroll field. After data has been entered on the last line of the field, the charges scroll up one line, again vacating a line for the next entry.

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.

Enter and Calculate. The scroll format illustrates another technique useful in format design. The operator enters only enough information on each of the charges to completely specify it. In our example, a charge is specified by the department and charge number. The system uses these to look up and calculate the rest of the information. It looks up the cost and the description. From these it calculates the new total and displays it immediately in its field at



SCROLL FIELDS



LOOK-UP AND UPDATE

the bottom of the screen. The date and time are also filled in by the system.

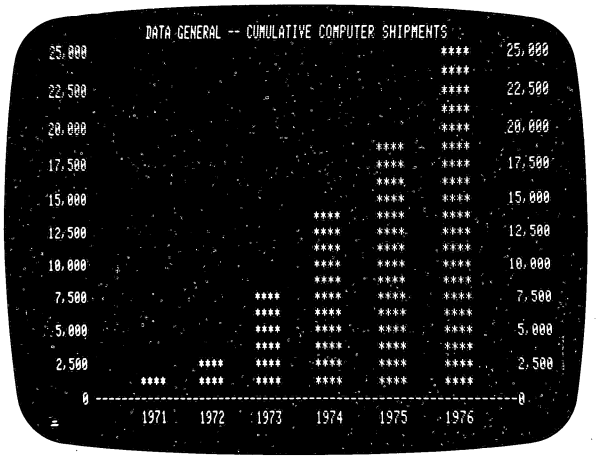
The advantages of this scheme: operator time is saved; the fedback information allows the operator to confirm the input, and calculation errors are eliminated.

Look-Up and Update. Another convenient format structure to use is the look-up and update format. The format can be used 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 data on the patient, as currently reflected, is displayed. The operator may then use it for reference or update it with new information.

Screen Overlays. In some applications, different kinds of data may be entered 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 the scroll and update examples share the same patient name and number and could utilize the overlay function. Here is how overlays are used.

The first format is displayed. One portion of it contains heading data. Another portion of it contains data on the first task. After the format is filled out, 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, the heading data does not need to be filled out again nor looked up in the database.

The overlay feature is quite versatile. The screen can be



BAR GRAPH FORMAT

divided in any proportion. An overlay format can consist of any number of disconnected parts. Parts can start and end anywhere, including in the middle of lines. As many overlay formats as desired can be used. Only one format, though, can be active at any given time.

Bar Graphs. Some data is best understood when presented visually. 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.

3. FORMAT STRUCTURE AND PREPARATION

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

Literals

Literals are used for the format title, to identify data fields and perhaps for brief instructions to the operator. Literals are protected; they cannot be changed from the operator's terminal keyboard. All keyboard characters are legal as literals.

Data Fields

The data fields are used for keyboard data entry and for data display. Each field has a defined format and length. The definition also specifies the class of characters legal for the field. The field can be restricted to numeric only, alpha only or alphanumeric characters. For numeric fields, the definition includes the decimal point location, can specify that leading zeros be suppressed and can include a floating arithmetic sign or a floating currency symbol.

Data Field Attributes

Each data field is assigned one or more of the following attributes—*Edit, Display, Output, Full, Required, Auto-Dup,* and *Auto-Entry.* Each allows the system designer to control data entry, display and storage for the field in a specific way.

Edit. An edit field is used to enter data into the applications program.

Display. Data generated by the applications program is displayed in this field.

A field may have both the *Display* and *Edit* attributes assigned. In that 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. Data entered into the field is automatically stored in the transaction file. The attribute *Output* can be assigned to display or edit fields. The data may originate from the applications program (*Display* and *Output*). Alternately, the data may originate from the keyboard (*Edit* and *Output* or *Output* only).

The following attributes can be assigned to edit or output fields. They cannot be used with display-only fields.

Full. A full field must be completely filled out or not filled out at all. The attribute is used for fields that have a constant number of characters, like a zip code or a telephone number.

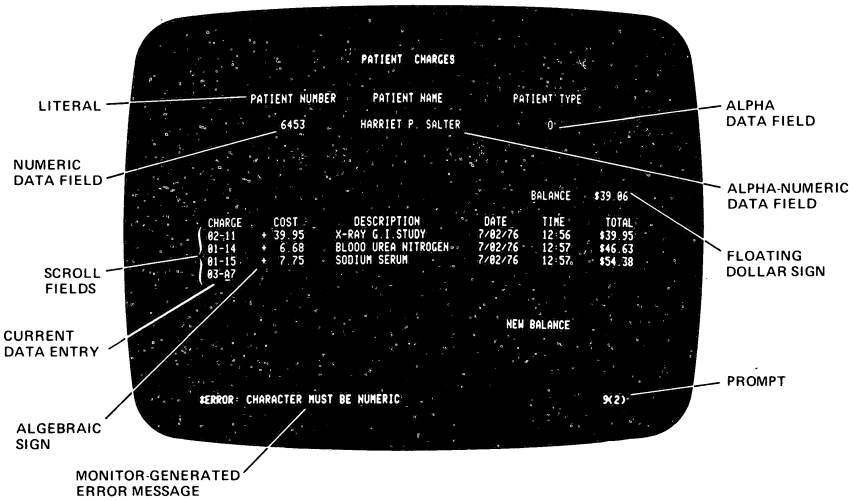
Required. A required field cannot be bypassed by the operator. Once it is encountered, an entry must be made in it. In addition, the operator is barred from exiting the format with the *Log Off* or *End of Data* commands until all required fields are filled.

A required field may also have the *Full* attribute.

Auto-Dup. To save keying effort, a field whose data repeats on every line in a scroll area does not have to be

FORMAT DURING DATA ENTRY

This is how the format looks during data entry. On the facing page, is the same format as it appears during preparation.



retyped each time. Such a field may be given the *Auto-Dup* attribute. Then it is typed once and automatically duplicated after that.

The *Auto-Dup* function is useful for output fields. 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.

Auto-Entry. Auto-Entry fields do not require striking the *Enter Field* key to terminate the field. Once all character positions have been filled, the *Enter Field* function is performed automatically.

Message Field

Error and information messages are displayed on the bottom line of the screen. They are generated by the monitor or by applications programs.

Creating Formats

Formats are created interactively, under the utility IFMT (pronounced I—format). IFMT can be run concurrently with the applications programs in the other ground. Here is how the format is prepared.

Placing the terminal in *literal mode*, the format designer types the literal data just as it is to appear to the operator. Editing functions allow the designer to correct errors and move the literals on the screen by inserting and deleting characters and lines.

The data fields are created by placing the terminal into *field mode*. Like the literals, the data fields are also typed in place. The designer 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 format. The printout is an exact character-by-character image of the format, showing the exact placement of all the elements on the screen. Data fields are indicated by the same mnemonics used to specify them.

The designer can switch from literal mode to field mode and back any number of times. The designer may move the elements of the format and change them repeatedly. When they are just as they are wanted, he/she puts the terminal into *assign attributes mode*. In this mode, the designer assigns each data field its attributes.

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

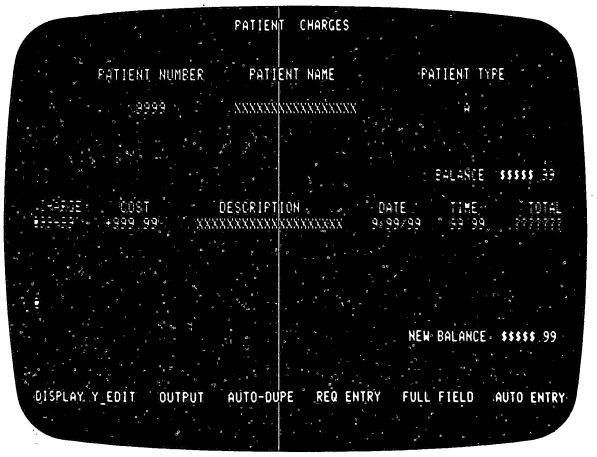
Hardcopy Printout Formats. Formats for printing reports are also generated under IFMT control. In printout mode, IFMT allows formats to be any number of lines in length. Otherwise print and display formats are identical. The same kind of structures are used. Thus reports consist of data fields and literals. The placement of all the elements is accomplished interactively on the screen, a much easier process than through programming. Similiarly, headings and other literals are typed directly rather than specified in a program.

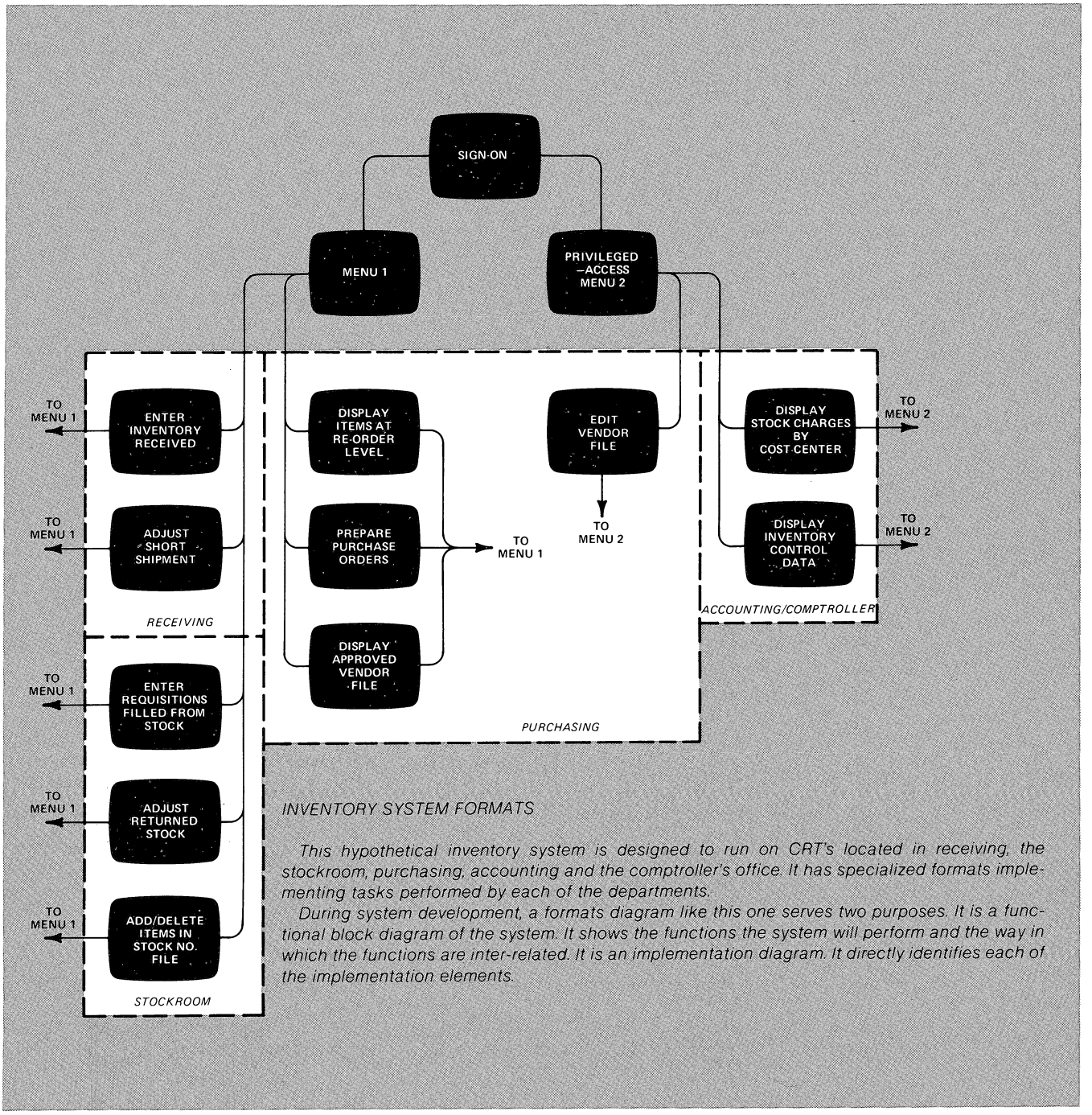
FORMAT IN PREPARATION

Headings and other literal information are typed in place, as they are to appear to the operator. Data fields, also typed in place are specified as alphanumeric (X), numeric (9), and alpha (A). Floating dollar signs, suppressed leading zeros and other features can also be specified.

After all elements are in position, data field attributes are assigned by answering the questions on the screen's bottom line.

Format preparation is then finished.





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.

4. ORGANIZATION OF APPLICATION SYSTEMS

An applications system in its fullest consists of all the formats, file structures and applications programs required to perform a function. For example, a function may be inventory control, accounts receivable, or hospital patient records.

Formats, the basic organizational units of Idea, correspond to data transactions. For example, the inventory system diagrammed 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 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 formats for business information. In short, there is a format tailored to every transaction and inquiry function the system is to fulfill.

Formats, Applications Programs

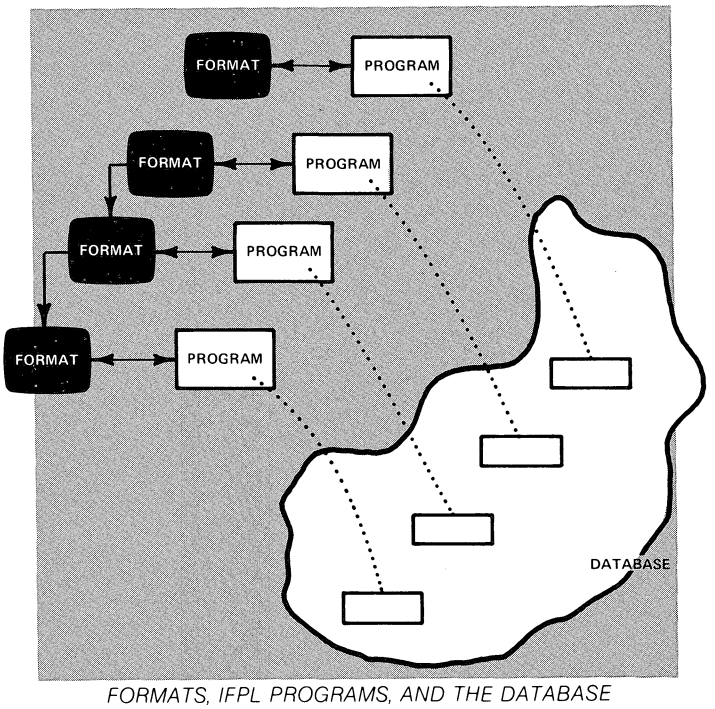
Formats may be supported by applications programs. These add considerable power to the system. They perform calculations on data, retrieve data from the database, and display data on the CRT, among many other things.

An applications program is permanently associated with particular format. When a format is called (at sign-on, for instance), its associated program is automatically loaded by the monitor.

Applications programs are written in IFPL, an acronym for Idea Field Programming Language. The name is suggested by the structure of IFPL programs. A program's structure closely parallels the structure of the format supported. Each format data field is processed by a specific procedure in its supporting program. The procedures and data fields are related to each other at compile time.

During processing, the monitor sequences through the format field by field. At each field, it gives the applications program control at that field's procedure. When processing of the procedure is completed, the program passes control back to the monitor. Because it is the monitor which takes care of the overhead of CRT communication and of keeping track of the current and next field, procedures in the applications program are short and direct. They must provide only for processing of the data and need not have any communications-related instructions. An example of a format and its procedures is shown in the box.

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



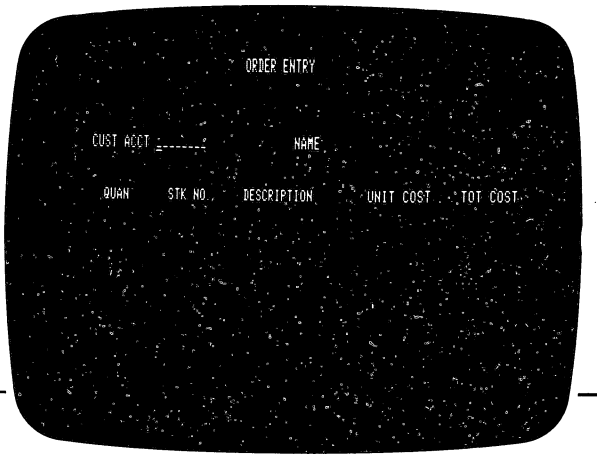
FORMATS, IFPL PROGRAMS, AND THE DATABASE

Formats are each supported by separate programs. They all have access to the same data records.

Other Monitor Functions

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

Error Checking. Error checking on entered data is performed automatically by the monitor. After each datum is entered, the monitor checks it against the criteria established during the format creation process. Checks include number of characters, class of characters (alpha, numeric or alphanumeric) and legal positioning of the decimal point in decimal numbers. If an error is detected, the operator must correct it before any further data is accepted.



SALES ORDER FORMAT

To understand the field orientation of IFPL and the natural way in which programs are written, it is best to consider a short example. Consider a sales order entry format. It might look like the one shown.

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 is necessary 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, for verification by the operator. The procedure looks as follows:

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

Using ACCT as the key, the customer record is found in the database. The customer's name, a variable within the record, is displayed. 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
```

The 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 variables DESCRIPTION and UNITPRICE.

```
DESC:   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
```

And that is the entire procedure section for the format shown. (In addition, there is also a definition section. See chapters six and seven.)

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, each module supporting a specific data field. Consequently it is an easy program to specify, to write and to modify.

Transaction File. The transaction file is a file into which the monitor automatically stores data from selected screen fields. The file can be used as a journal or audit trail. It can be used to transfer data to other systems.

The monitor automatically creates records tailored to the data input requirements of each format. As data is entered into the format, it is automatically filed in the database. The data records are indexed by CRT number, batch number, and format type. Date, time and operator ID are also recorded.

The transaction file is in addition to any files created by the user. IFPL programs explicitly store and read data in these files.

Linking Formats. Sometimes it is convenient to implement a particular task using two or more formats. Formats can be linked, so that when one format is completed the next format required for the task is automatically displayed.

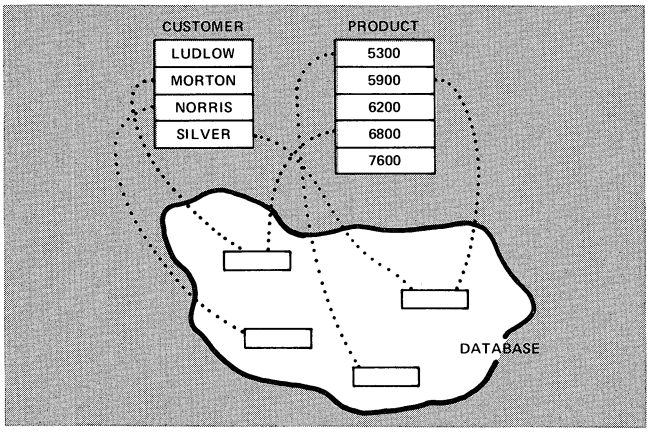
Format-Only Systems

The monitor performs the above functions—error checking, transaction file entry, and format linking—entirely from parameters specified in the format generation process. Criteria for error checking is derived from the field definitions, the transaction file records are created directly from the data fields, and formats are linked as specified by the format designer. Thus these functions require no program support. In fact simple data entry formats can be run entirely without a program. Data entered into formats of this type is checked by the monitor and, after validation, is stored in the transaction file.

5. INFOS DATABASE MANAGER

The Idea system uses Data General's powerful INFOS data manager. It is implemented at its highest level, DBAM, for DataBase Access Method.

Under INFOS, a database is truly just that—a pool of data accessible to various programs each for its own purpose. Consider a database containing sales orders. Under INFOS, it is natural and convenient to use the same database for billing, for calculating sales commissions and for statistical management reports, to name just three applications.



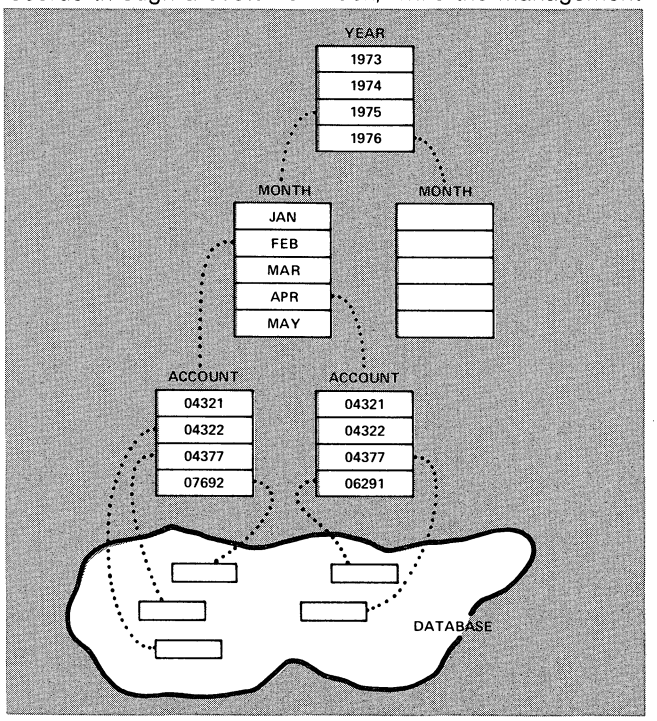
(a) Records can be accessed via two or more indexes.

The programs accessing the database do not all have to be written in the same language nor all operate 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, the billing and sales commission programs might be written in Cobol, RPG or another language and run in batch, in another ground.

Labor is clearly saved because the data is input and updated only once for any number of applications. But though this may be significant in many cases, it may not be the most important benefit of INFOS. Under INFOS, the database becomes a growing corporate resource, ready to be used in the future in ways not envisioned when the data was collected.

INFOS Structure

The ability to access the same database differently for different purposes is inherent in the way INFOS structures data. The database consists primarily of records and indexes. Each record need be stored only once. But it may be accessed by any of a number of indexes by attributes. Thus, it is available for different purposes. To continue the example, the billing program would access sales order records through a customer index, while the management report program might access them through a product index (a).

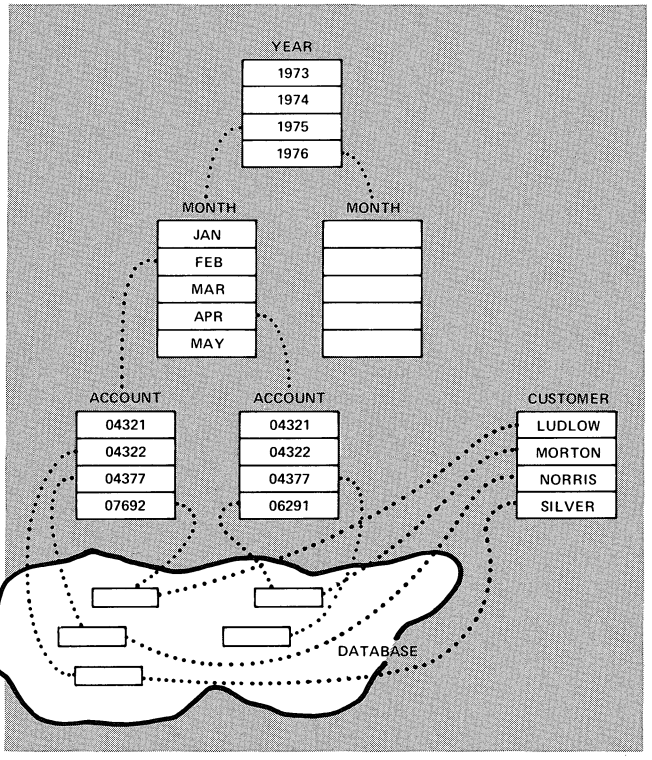


(b) Multi-level Indexes

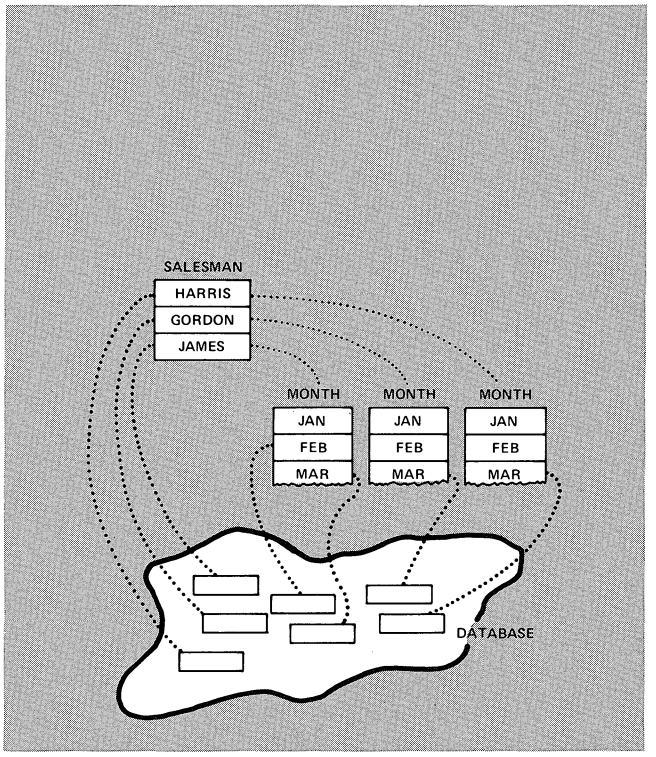
An indexing structure may be multi-level (b), having an index and several levels of subindexes. (see note 1) Thus, hierarchical data structures are easily made. For instance, sales records might be organized by year, month and

NOTE 1

The number of indexes and subindexes is not limited by INFOS. However, each IFPL applications program is restricted to opening a maximum of three files (indexes) sharing a total of 15 indexes and subindexes. This can be apportioned in any way. At one extreme, IFPL program might handle a single file with an index and 14 levels of subindexing. The transaction file, the print file and the common file and their indexes and subindexes are not included in this limit. They are automatically available in addition.



(c) Separate index paths can have different numbers of levels.



(d) Index keys can access records directly as well as through subordinate subindex levels.

account. In such a structure, year keys (entries in the year index) point to month subindexes. 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.

Of course, not all indexing paths need have the same number of levels (c). Records might be accessed along one path through only a single index and accessed along another path through a dozen levels. In fact, INFOS is quite flexible in the types of index structures and records that can be used. Many variations of the basic index structure are allowed. Each extends the usefulness and convenience of the database. For instance:

An index or subindex may reference subordinate subindex levels as well as reference records directly (d). For instance, sales orders would be accessed via salesman-year-month path while records containing salesman expense reports might be accessed by the salesman index directly.

Any number of keys within a single index can point to the same record or subindex (e).

Duplicate keys are allowed at the lowest index level (f). They are distinguished from each other by an automatically-generated occurrence count. The occurrence count 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 Smiths added to the database are given the next highest occurrence count. Occurrence counts used by deleted keys are not reassigned.

Keys within an index can be of mixed type and can be of different lengths (g). Keys may be compressed (h). Access via full keys, generic (also known as *partial)* keys and approximate keys is supported (i).

Different length records containing different types of information can be accessed by the same index path. Records may be locked on an individual basis. Thus simultaneous access to the same record by different, unsynchronized users is prevented. Yet all users can have simultaneous access to all other parts of the same files.

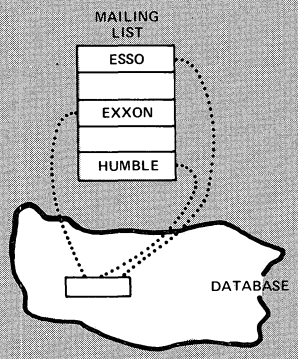
Creating Database Structures

The database structures described above are created, modified and deleted by a series of interactive INFOS utilities. The utilities engage the database designer in a question-and-answer dialogue on the system CRT. From the answers, they create or modify the database. No programming is required. However, designing large databases to use storage efficiently and to have good access time does require specialized knowledge.

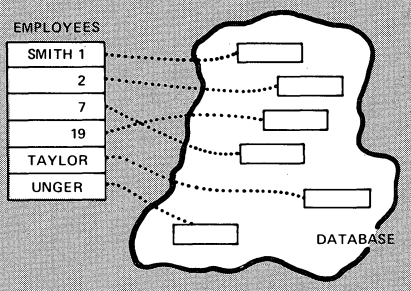
The INFOS utilities commonly used in database maintenance are

ICREATE	IDELETE
INDEXCALC	IRENAME
ICOPY	INQUIRE

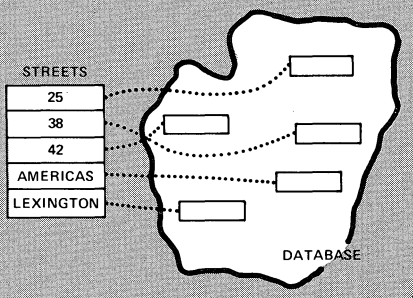
The utilities are described in the *INFOS Utilities User's Manual.* This manual and other INFOS documentation is listed in the Preface.



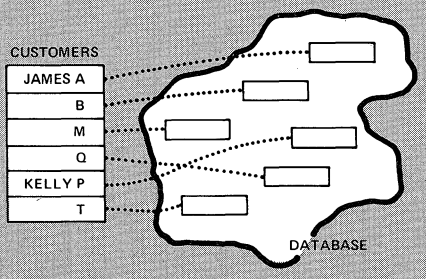
(e) Any number of keys within an index can point to the same record.



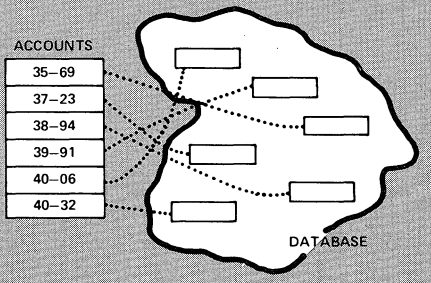
(f) Duplicate keys are allowed. They are distinguished from each other by an occurrence count assigned to each key automatically by INFOS.



(g) Mixed Keys
A single index can have keys of different data type and length.



(h) Compressed Keys
To save storage space, INFOS stores only the unique portion of a key. The redundant portion is not repeated.



(i) Generic Keys; Approximate Keys
The generic key 37 accesses the record stored under key 37-23. The approximate key 3800 accesses the first record following that key position; in this case 3894.

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. Auxilliary words like *to, for* and *is* can be included within statements to make programs more readable.

The IFPL language is described in three sections. This section treats most processing statements. File handling statements are described in the next section. Section 8 concludes the IFPL description with a discussion of the structure of IFPL programs and details on their interaction with the IDEA monitor, IMON.

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

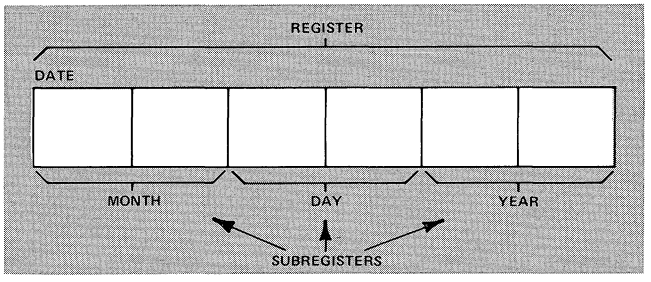
Definition Statements

Defining Storage Requirements

Memory storage is allocated by the REGISTER statement. The amount of storage reserved, the datatype, and the position of the decimal point for numeric variables is specified by a Cobol-like picture statement.

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 and yet may need to be treated separately. Similiarly, a check digit may need to be separated from an account number.



SUBREGISTERS

The elements of concatenated data are generally cumbersome to handle. They are handled easily, however, by dividing the variable into subregisters. The resulting subregisters behave as any other variables.

These requirements and others like them are conveniently handled by the REDESIGNATE command. This command assigns a name to specified portions of a register. For instance, the first two characters of the register Time might be assigned the name Hour, the second two the name Minutes and the third two the name Seconds. The entire register Time and each of its sub-registers, Hours, Minutes, and Seconds behave as independently addressable variables. Values can be moved from and into them, they can be displayed, used as elements of tables and used in calculations.

A register can be divided into any number of sub-registers. Sub-registers may overlap. Not all characters of a register must be included.

Defining Data Field Processing Modules

As more fully described in the section *IFPL Structure*, each data field is processed by a particular IFPL program module. The module is specified by a PROCESS statement. To save programming effort, the PROCESS statement also reserves memory storage space for the data, obviating the need for a separate REGISTER statement for storing screen data fields. The definition of the data field made during format preparation serves as the picture statement.

Creating Tables

Tables are created by the TABLE statement. The table entries are simply typed in their proper order. Entries can be literals or registers. The table is terminated with an ENDTABLE statement.

Executable Statements

Communicating with the CRT

The IFPL language has special statements for convenient communication with the CRT display. Data entered by the terminal operator is read from the screen and stored in a specified location by the STORE statement. Data is displayed in a CRT data field from any memory location by the DISPLAY statement. Error messages and instructions are displayed on the CRT by the MESSAGE statement. In addition, a complete, new format can be displayed by the LINK instruction. (See *Linking to Other Formats*, below.)

Branches

IFPL provides both conditional and unconditional branch statements. Conditional branches are written as statement pairs, consisting of a test instruction followed by an instruction specifying the branch condition and location.

Compare. The COMPARE statement is associated with one or more of the following branch conditions—IF EQUAL, IF NOT EQUAL, GREATER THAN and LESS THAN.

Range. The RANGE statement tests the 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 whatever statement follows OUT-RANGE is executed.

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

Lookup. The LOOKUP statement, treated further under *Tables*, also may be followed by branching statements. LOOKUP searches for a variable in a particular table. It may be followed by the branch conditions IF FOUND or IF NOT FOUND. It may be used, for instance, to check the validity of a password.

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

In the 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 branches to the routine PROCEED. Otherwise a message is displayed and normal processing does not occur.

On-I/O Errors. A quite convenient branch instruction is ON-IOERR. It is used after any of the database access instructions to branch to the I/O error handling routine. It allows I/O errors to be handled with the minimum of extra instructions within the mainline of the program.

Go To. 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 writing routines where the branching location is dynamically determined by the program. An example is shown under *Tables,* below.

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

Subroutines

The IFPL language accomodates subroutines for common processing. The PERFORM statement branches to the subroutine named in its argument. The subroutine entry point is defined by a SUBROUTINE statement and its exit with an ENDSUB statement. When ENDSUB is encountered, processing branches back to the instruction following the subroutine call. Subroutines may call other subroutines. There is no limit to the number of nested levels. 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. The basic scheme employed is this:

A table and all its elements are predefined in the definition (non-executable) portion of the IFPL program. Table elements are addressed as TABLENAME(INDEX), where INDEX is the register containing the element's positon in the table. This table address may be used as the argument to the MOVE and DISPLAY commands. Thus variables can be moved from or into tables and displayed directly from tables.

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

Up to ten tables may be defined. Each table may have up to 99 elements.

A variable is looked up in the table with the LOOKUP instruction. If it is found, the system stores its table position

in the register specified. The table position may be used to index a variable in a second table. It can also be used 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 a six error states, identified by non-consecutive number codes. In the example, these error codes have been defined as elements of the table ERRTABLE. The reserved word IOERR contains the error code. The routine for determining which error occurred and to where, consequently, the program must branch, might look like this:

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

IOERR's position in the table ERRTABLE is used to select one of the six branches, E10 . . .E96, specified in the 'GO TO USING' statement.

Computation

IFPL has a complete fixed point arithmetic capability. Computation is performed by the ADD, SUBTRACT, MULTIPLY and DIVIDE instructions. The result's accuracy depends on the size of the receiving variable. The variable can be up to eighteen digits long. The decimal point can be located anywhere within the variable.

Linking to Other Formats

An IFPL program can link to another format and its associated program. 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 statement can link to the correct format directly from the operator's input. A routine to process a menu format is shown in the box.

Variables can be passed from an active program to its successors. The data is passed via a special system file. 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 a particular function can be implemented by two, three or more consecutive IFPL programs. Intermediate data can be passed from program to program, including from the first program to the last.

In a menu, an operator is asked to pick one of a list of alternatives by keying its item number. The item number is used 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 chosen format, indicated by ITEM, is moved to the temporary register FORMAT, and the link is performed.

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

The RETURN statement returns control to IMON, the monitor, which displays the new format and loads its applications program.

This routine together with a process statement and some register statements is the entire program for processing a menu. The four statements above are the only active statements required.

Source Library Files

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

All standard program modules are written once and stored as individual source files. They are inserted into any IFPL programs using them by a single statement—COPY. The IFPL compiler expands the COPY statement to the full contents of the referenced file.

Generating Reports

IFPL programs can generate data for reports.

To reduce the overhead in the applications program, the report printouts use formats just like the CRT displays. The formats are created interactively, under IFMT in the same way. Headings, data identifications and other literal information, as well as the definition and page placement of the data fields are all made under IFMT. Thus none of these functions need be performed by the applications program.

All that must be done in the IFPL program is the filing of data into a special system file. The data is filed in the order required by the format. Later, a utility run in another ground or off-line reads the file and maps the data into the format as it prints it out.

Storing Data in the Print Format. Conceptually, the print file consists of a serial string of printouts. Each printout contains a data record or records matching the requirements of the format for that printout. The printout's beginning and end in the file are marked. These marks are later used by the print utility as it maps the data into the print format.

Three IFPL commands are used to generate the print file. They are:

Initiate Printing. This command inserts the start-of-printout mark into the print file.

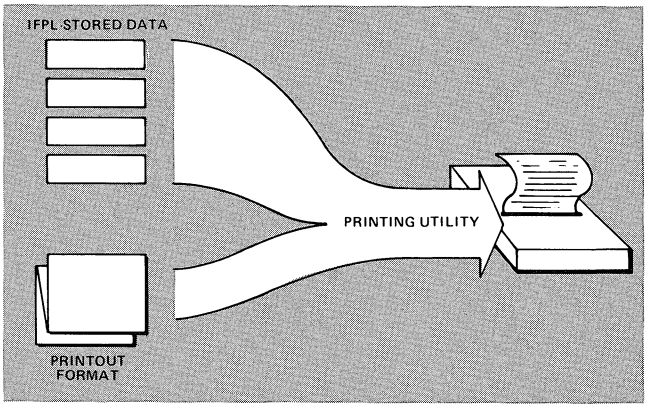
Print. This command stores a data record in the print file. A record corresponds to each page area (area between tabular areas) of the printout format or to each tabular line of the printout.

Terminate Printing. This command marks the end of the printout.

All three of the above commands reference a particular printout format. Thus the program can accumulate data for several printouts concurrently. Each printout would have to be initiated and terminated separately.

Mediating Operator Keyboard Commands

The operator keyboard commands *Backtab, End of Data, Escape* and *Log Off* usually require no code in the applications program (Note 1). They are handled by the monitor. However, there are cases in which a program must control the use of these commands. For example, a program might disable the *End of Data* command when it locks a record to prevent the operator exiting the program before the record is released (Note 2). Similarly, other commands may be disabled or their operation altered during all or part of a program.



REPORTS

Reports are printed by merging data stored by IFPL programs in the printout file with literal and position information established by the printout format. The formats are created interactively using the same utility used for screen format preparation.

The commands are disabled or modified in the following manner:

An IFPL routine is defined for each command that is to be modified. When the command key is struck by the operator, the monitor does not execute the command but rather enters the IFPL program at the specified routine. The processing for the command is whatever is contained in the routine. It may include displaying an error message and performing required housekeeping chores. The routine may follow different lines of processing at different stages of the program. The number of the last data field accessed by the monitor is available to the program in the reserved word FIELD.

NOTE 1

The *Escape* key differs from other operator commands in one respect: Its function must be implemented by the applications program. There is no monitor action defined for it.

NOTE 2

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

7. IFPL FILE HANDLING STATEMENTS

The database structure and access capabilities of INFOS are available to the applications program through a set of IFPL statements. The statements are of two types—definition statements and operation statements. The definition statements specify the file structure to the IFPL program (see note 1). The operation statements store, retrieve and delete data.

Database Definition

An INFOS structure is fully characterized by four definitions—the files that are to be accessed, the index paths, key lengths and record parameters.

Files. The files the program will use must be identified in a FILE statement (note 2). An IFPL program may open up to three files. The maximum number of index and subindex levels that these files can use is fifteen. The fifteen levels can be apportioned among the three files in any way.

The restriction on the number of files and index levels does not include system files. Thus, in addition to any files the program expressly opens, the program can use the system maintained transaction and common files.

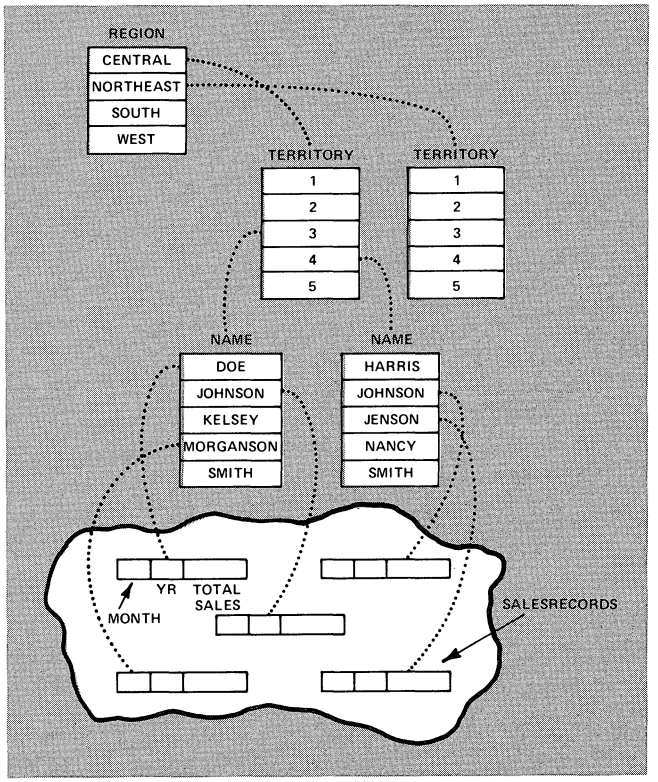
Indexing Path. The indexing path required to reach each record type must be specified. This is done by a statement for each subindex level. Thus, a three-level salesman file indexed by region, territory and salesman's name would be specified by two statements:

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

These two statements establish the index chain—region is linked to territory; territory is linked to name.

Key Length; Duplicate Keys. For each index or subindex level, there must be a KEY statement. The statement specifies the maximum length of the keys contained in the index. INFOS uses this definition to create subindexes as they are required.

The system allows keys at the lowest level to be duplicated, (see INFOS section) at the option of the program. If duplicates are to be allowed, a register for storing the duplicate occurence count must be defined. The statement is DUPLICATES COUNTED.



THREE-LEVEL SALESMAN FILE

Salesmen's sales records are accessed by region, territory and salesman's name. The record is packed; it contains month, year and total sales in a predefined format. See text for a discussion of the IFPL statements used to define the file structure and the record format.

Record Description. Next, a record type must be associated with its index or subindex. One of the records the file contains is of monthly sales. The statement

```
RECORD FOR NAME IS SALES
```

establishes this.

NOTE 1
The files are not created in the IFPL program. They must already exist. Files are created under the control of the INFOS utility ICREATE.

NOTE 2
The IFPL language uses the word 'file' synonymously with 'index'. Thus, the file name is the name of the highest level index.

File Retrieval Statements

Access Mode	IFPL Statement	Definition
Full keyed access	FIND	Retrieves record exactly as specified.
Generic key	FIND BEGINNING	Retrieves record the beginning of whose key is given. (note 4)
Approximate key access	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.

Each record type has a fixed format—and contains a specific number of variables arranged in a predefined sequence (note 3). The variables are packed, they are not separated by delimiters. Thus a template is required 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
```

Its length and constituent parts are specified by 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, for instance, specifies that the variable SALES starts at the fifth character, and is seven ASCII characters long. From the includes statements we can determine the month is 11, the year is 76 and sales is 46,057 in the record above.

Cobol programmers will appreciate the following labor-saving feature. Only the variables in a record actually used in the program require the INCLUDES clause. Variables not used need not be listed. Thus if the SALES record in the example included other variables that were not used in the program—it could conceivably have dozens more—the record statement shown above would be no longer. Considerable nuisance typing is saved.

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

To use these additional records, the index path definition statements need not be repeated. Only the additional records need be defined.

Data Retrieval

Variables stored on the database are retrieved by a FIND statement. The FIND 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 FIND statement is executed, the variables within the record can be manipulated like any others. The INCLUDES definition is all that is needed to define their names and positions within the record. The variables can be moved to other memory locations, displayed on the CRT, and used in calculations.

All the access modes discussed in INFOS are implemented with versions of the FIND statement. These modes, their FIND statements and definitions are listed above.

Verify. In some circumstances, the record itself is not required. All that must be determined is if 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 is intended for cases of this kind. VERIFY checks 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.

Record Locking. If the accessed record must be protected from change or access by other programs operating at the same time, the record may be locked. Each of the FIND commands 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. A parallel syntax is employed for the others.

The record is automatically unlocked when it is refiled, (see below). Alternately, the record can be explicitly unlocked by a RELEASE statement.

NOTE 3

Record variables are also sometimes known as fields. This terminology has been avoided here so there would be no confusion with screen data fields.

Once a record is accessed the variables within it can be manipulated just like all other program variables. It makes no difference if a variable is declared in a register, calculated or input from the CRT or read from the database record.

NOTE 4

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

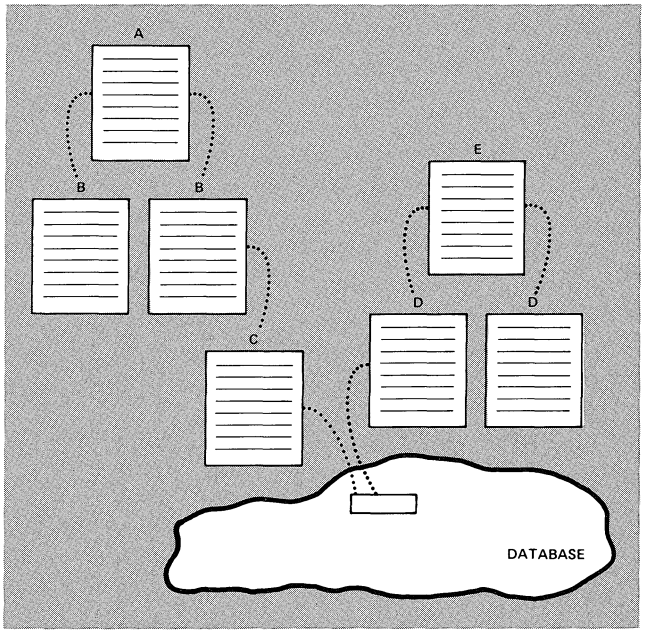
Data Storage

Variables are stored on the database by reversing the operations required to access them. The record is first assembled in processor memory. This is done by specifying the values of all record variables that are to be changed. Then the record is stored 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. Arguments following both instructions specify the record type and the keys by which it is filed.

Inversion. Each time a new record is filed into a cross-indexed structure it must be inverted through all other index paths.

Consider the cross-indexed structure shown in the figure. In this structure, each record can be accessed either via the index path A,B,C or via the path D,E. When a new record is stored via one index path, for instance A,B,C, it must be *inverted* through the other path, D,E. Inversion consists of updating the keys in the second path so that the record can later be accessed by that path. The IFPL command is INVERT followed by record and key arguments.



INVERSION

A record stored via the path A,B,C must be inverted via path D,E so that it may be retrieved via either path in the future.

Deleting Records

Records are deleted from the database by one of two commands.

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

The REMOVE command performs a logical delete. In effect, it flags the record as deleted. The record is still accessible. When it is accessed the reserved word IOERR is set to 96 to indicate the record has been logically deleted.

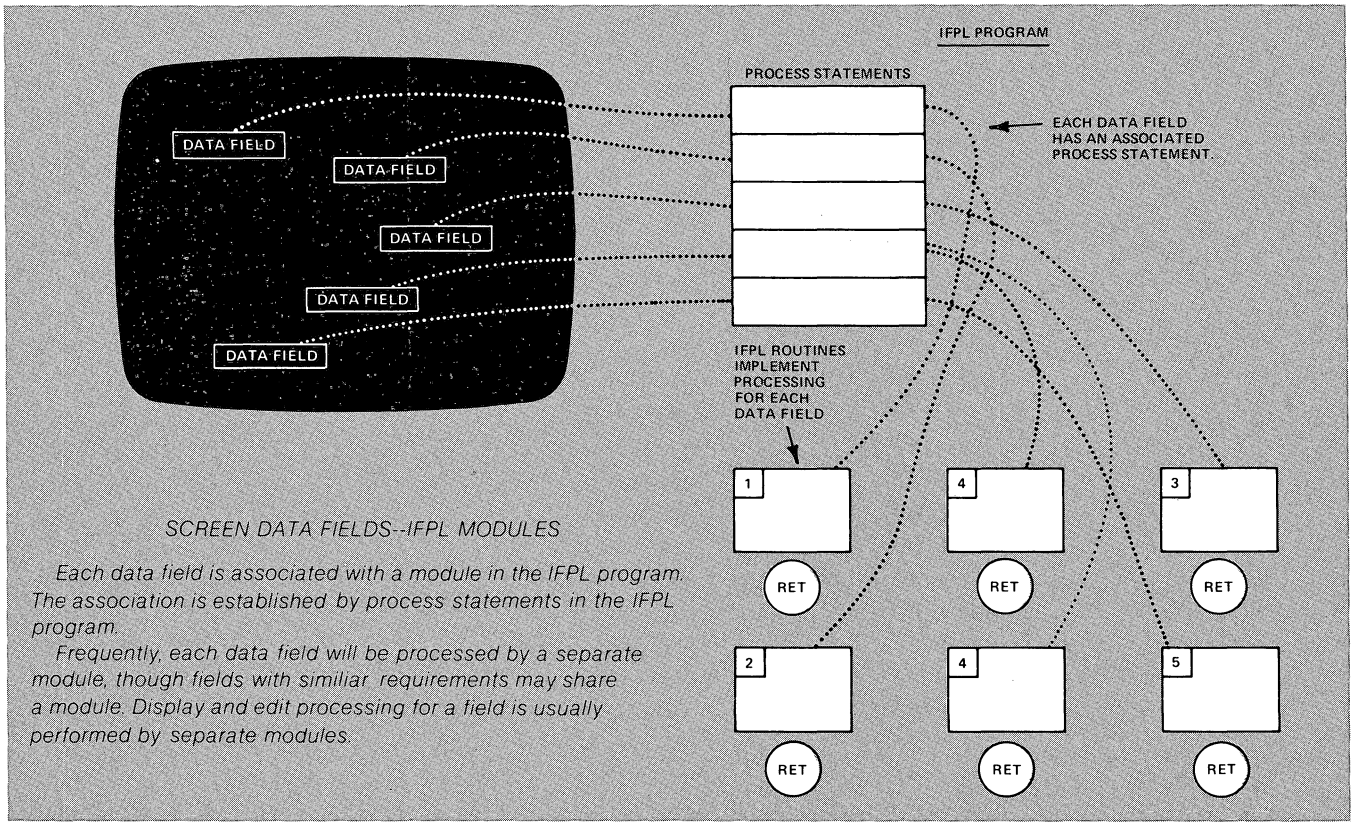
The REMOVE command, though it does not actually perform a deletion and free disk space, might be used in preference to the DESTROY command for two reasons. It is safer than the DESTROY command; its action is reversible. It is faster than the DESTROY command. Thus, the REMOVE command might be used by most IFPL programs. Periodically, all logically deleted records might be physically deleted by a separate program.

These commands, like the other record reference commands are followed by record type and key arguments.

8. STRUCTURE OF IFPL PROGRAMS

Each IFPL program supports a specific screen format. Much of the ease and speed with which IFPL programs can be written depend on the close corresponence between IFPL structures and analagous format structures. The program relates to the format in three principle ways:

Monitor Interaction. Control is passed from the monitor to the next IFPL program module and back to the monitor on a field-by-field basis. Overhead associated with CRT communications is thus directly handled by the monitor. It requires no support from the program.



SCREEN DATA FIELDS--IFPL MODULES

Each data field is associated with a module in the IFPL program. The association is established by process statements in the IFPL program.

Frequently, each data field will be processed by a separate module, though fields with similiar requirements may share a module. Display and edit processing for a field is usually performed by separate modules.

Field-Orientation. Each data field in a screen format is permanently associated with an IFPL program module. As a result, IFPL programs are naturally modular. A module, usually quite autonomous, simply is the procedure for handling a particular field's data. Often, modules can be written and later modified independently of the rest of the program. (see note 1)

Processing Flow. Each time the monitor gains control, it positions cursor at the next logical field and re-enters the IFPL program at the module specified for that field. Flow is field-determined.

Each of these aspects of IFPL programs is discussed more fully below.

Note 1

If a data field has no 'Edit' or 'Display' attribute, it has no corresponding module in the format's IFPL program. From the standpoint of the program, processing is as if the field did not exist.

Though a data field is not included in the program, it is recognized by the monitor. The monitor will position the cursor to the field, check the input for errors. The monitor will file the data in the transaction file if the 'Output' attribute were specified for the field.

Field-Orientation

IFPL programs are *field-oriented*—that is, each screen format data field is associated with a processing module within the IFPL program. Each field may have its own, unique module. 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 similiar processing requirements may share a single module.

The processing module (or modules, for display and edit fields) is assigned to a data field by a PROCESS statement.

Monitor Interaction

During processing, control is interchanged between the IFPL program and the monitor on a field by field basis.

Here is how a single data field is processed.

The monitor moves the cursor to the data field.

If the field is an edit field, the monitor waits for the operator to enter the required data and checks the data for errors. After the error checks are satisfied, the monitor stores the entered data in a reserved area. From there, it is available to the IFPL program by the STORE command. Then the monitor passes control to the IFPL program module specified for that data field in that field's process statement.

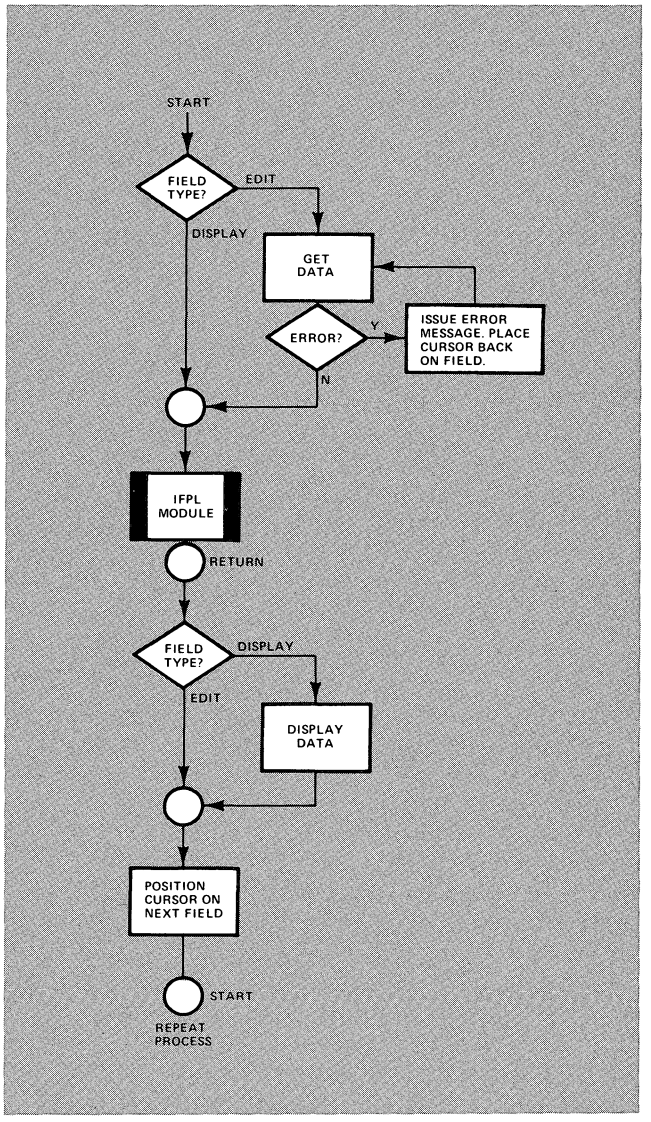
If the field is a display field, control is immediately passed to the specified IFPL program module. Actual display of data in the field does not take place until after execution of the IFPL module.

Processing now proceeds 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. It is important to note that any STORE or DISPLAY instructions are interpreted to pertain to the current field regardless of where they occur within the IFPL program. This is true even if the program branches to code within a module associated with another field.

Control passes from the IFPL program back to the monitor when a RETURN statement is encountered. If the field were a display field, the monitor displays any information specified by the IFPL program. The monitor then positions the cursor at the next logical data field and the process repeats.

Processing Flow

Processing flow proceeds from one field and its associated module to the next logical field and its module. Thus,



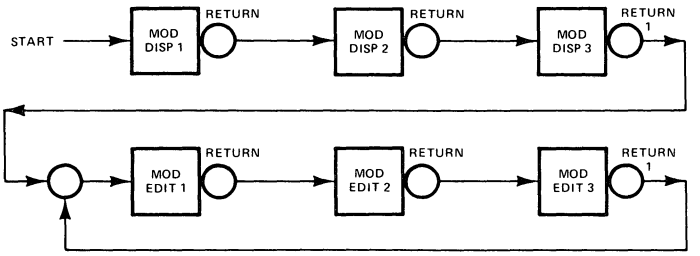
PROCESSING OF A SINGLE DATA FIELD

Monitor and IFPL program interaction in the processing of a single data field is shown in the flow diagram. The monitor performs the terminal communications overhead functions. The IFPL module manipulates the data.

After IFPL processing of a field is completed, the monitor once again assumes control. It positions the cursor to the next field, validates keyboard data and re-enters the IFPL program at the module defined for that 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.



there are two distinct aspects to determining processing flow—determining which is the next logical field and then determining which module (fields with both *display* and *edit* attributes are associated with two modules) is effective. These aspects are discussed separately below.

The Next Logical Field.

Generally, the next logical field is the next physical field on the screen. The data field sequence is 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. The next logical field may be specified in the RETURN statement. The monitor will position the cursor to the specified field and branch to its effective IFPL module.

This form of the RETURN statement is used to exit loops under program control. It is the proper 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 a scroll area is the first field of that scroll area. Thus, a scroll area creates an implicit loop in the program. The loop must be explicitly exited by the IFPL program or by an operator command if data fields outside the scroll area are to be processed after processing within the scroll area has started.

Display and Edit Processing. The first determinant of execution order is the next logical field, as outlined above. After the monitor determines the field, it enters the program at the effective module specified in that field's PROCESS statement. If only one module is specified, that module is processed every time the field is encountered. If two modules are specified, one for display and one for edit processing, the effective module is determined by the following rule:

The first time the field is encountered, processing is at the display module.

The second and every subsequent time the field is encountered processing is at the edit module. (see note 2)

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

Starting Address. Implicit in the processing order described above, is the starting location of the IFPL program. The program will start at the effective module of the first screen field. Occasionally, however, there may be some initialization or other processing that must take place before the first field. In that case, the convenient expedient is to create a single character display-only field at the beginning of the screen. Its associated module can be used for initialization or whatever.

At the End. Where does processing flow after the last module is executed?

If the module for the last field ends with just a RETURN (a RETURN without a field number) the monitor links to a new format—the format specified during preparation of the current format. (Specifically, the format entered in answer to the question 'Link?' asked at the end of IFMT.) This facility can be used to link to the next task or to the system's ground state.

If no format were specified in response to the 'Link?' question, control is returned to the monitor. The monitor flushes the screen and displays either the ground state format or the question 'Format?' to the operator.

Other Considerations

Because processing flow is usually determined by field order in the screen format, arrangement of data fields is quite important. If data fields are arranged in the order information is available, as for instance the blanks are in the 1040 income tax form, the supporting IFPL program is easy to write. Conversely, if an IFPL program is difficult to write, rearrangement of data fields to a more logical order will frequently help.

The GO TO and the RETURN statement are sometimes confused because both statements cause processing to branch to another portion of the IFPL program. The GO TO

Note 2

A scroll field with both the 'Display' and 'Edit' attribute is processed as follows.

The first time the field is encountered, it is processed by the display module. If the field is encountered again, on the same line, it is processed by the edit module. Once the next scroll line is entered, the field is treated as if i were being encountered for the first time. It is again processed first by the display module, and if encountered again, by the edit module.

branches to the location specified by its argument. The RETURN branches to the location specified in the next logical PROCESS statement.

However, there is a crucial difference between the two. The GO TO branches immediately. No matter where it branches, the current screen field remains the same. All DISPLAY and STORE commands are still interpreted with respect to the same field.

The RETURN statement, on the other hand, branches via the monitor. When the IFPL program regains control, the effective data field is the next logical field. All DISPLAY and STORE statements are now interpreted with respect to a new data field.

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 transaction driven and interactive. The database is on-line. A number of operating features make the system particularly easy to learn and to operate. These include error detection, prompting, and dedicated function keys on the keyboard. In addition, the applications software can easily incorporate menus and mther interactive operator aids. As a result data entry and retrieval fuctions can frequently be moved to operating departments with attendant improvements in responsiveness, reduced error rates and cost savings.

Formats

System data transactions are expressed as screen formats. As many formats as required may be created. Formats can link to other formats automatically, either unconditionally or as a result of operator action.

Formats are prepared interactively under the Idea utility, IFMT. Essentially, the format literals and data fields are typed as they are to appear to the operator.

Data fields can be specified as Alphabetic, Alphanumeric or Numeric. Numeric fields may have a floating currency symbol, an arithmetic sign, a decimal point, leading zeros supressed, and a check-protect character. Data fields are assigned one or more of the following attributes: Display, Edit, Output, Required, Full, Auto-Dup and Auto-Entry.

INFOS Database Manager

Idea uses INFOS DBAM database structures. Data is stored in records retrieved via indexes. The indexes may be multi-level. Data records may be cross-indexed. Retrieval is by key. In addition to full keys, approximate keys, and generic keys may be used. Sequential access in both the forward and reverse directions is also supported.

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

Applications Programs

Applications programs are written in Cobol-like 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 IMON, the runtime monitor, to perform many of the overhead functions required for interactive screen I/O.

IFPL STATEMENTS

Definition
- REGISTER
- REDESIGNATE
- PROCESS
- TAB LE
- ENDTABLE

Communicating with CRT
- STORE
- DISPLAY
- MESSAGE
- LINK

Subroutines
- PERFO RM
- SUBROUTINE
- ENDSUB

Conditional Branches
- COM PARE
  - IF EQUAL
  - IF NOT EQUAL
  - GREATER THAN
  - LESS THAN
- RANGE
  - OUT—RANGE
  - IN—RANGE
- LOOKUP
  - IF FOUND
  - IF NOT FOUND
- ON I/O ERROR

Unconditional Branches
- GO TO
- GO TO USING
- RETURN
- RETURN USING

Linking
- LINK USING
- PASS
- ACCEPT

Computation
- ADD
- SUBTRACT
- MU LTIPLY
- DIVIDE

File Definition
- RECORD
  - LENGTH
  - INCLUDES
  - REDEFINES
- F ILE
- SUBINDEX

File Retrieval
- FIND
- FIND NEXT
- FIND PREVIOUS
- FIND BEGIN—NING
- VERIFY

File Storage
- REFILE
- FILE—NEW
- DESTROY
- REMOVE

Record Locking
- HOLD
- RELEASE

Hardcopy
- INI TIATE PRINTING
- PRINT
- TERMINATE PRINTING

Mediating Keyboard Commands
- ON BACKTAB
- ON END OF DATA
- ON ESCAPE
- ON LOG OFF

Other Statements
- MOVE
- COPY

Appendix 1
MEMORY REQUIREMENTS

Idea Release 1.0

Approximate memory required to support each of the components of an Eclipse system running the Idea software package is listed below. The list is intended for planning purposes and applies only to the software release numbers shown.

	k bytes
RDOS/INFOS (note 1)	64.0
IMON, including buffer space for 2 terminals	56.0
Additional terminal buffer space (note 2)	
per additional terminal -- minimum	2.5
or	4.5
or maximum	6.5
File buffer space	
for system file Common	0.75
for system file Trans (note 3)	0.75
for each user file opened concurrently - minimum required (note 4)	0.75
Background processing	
Maximum required (for Idea Compiler)	64.0

Notes.

1. RDOS release 6.0; INFOS release 2.0.

2. Terminals can be allocated one, two or three 2k byte blocks of memory for IFPL program storage. Maximum program length is limited to the amount of memory selected. An additional 0.5 k bytes of memory per CRT are required for storage of format data field definitions. Total CRT memory space must be rounded up to the next highest integral thousand bytes.

Actual allocation of memory is performed when IMON is started up. Memory is allocated only to the number of terminals selected at that time. All other memory is available for other-ground tasks.

3. Trans file memory is not allocated if transaction output is supressed. This may be done at IMON start-up.

4. 0.75 k bytes per file consists of two 0.25 k buffers for the file index, and one 0.25k buffer for the database of each open file. For faster response time more memory should be allocated.

DataGeneral

REMARKS FORM

Document Title	Document No.

SPECIFIC COMMENTS: List specific comments. Reference page numbers when applicable. Label each comment as an addition, deletion, change or error if applicable.

GENERAL COMMENTS: Also, suggestions for improvement of the Publication.

FROM:

Name	Title	Date

Company Name

Address (No. & Street)	City	State	Zip Code

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

Att. SYSTEMS PRODUCTS DOCUMENTATION

Mail Stop 7-33

FOLD UP SECOND FOLD UP

STAPLE