INFOFLEX®, INC

I N F O F L E X - 4 G L

User Guide

Printed in U.S.A. on May 2006

TABLE OF CONTENTS

1. INTRODUCTION TO INFOFLEX

What Is Infoflex?

InfoFlex is a complete 4GL application development system that will meet the needs of both the non-technical user and the most serious vertical market developer. **InfoFlex** may also be used to enhance existing applications written in Informix, SCO Integra, D-Isam, C-Isam, Micro Focus Cobol, or SUN Netisam because of their compatible file structure.

Infoflex includes packages for developing menus (MENUFLEX), data entry screens (SCREENFLEX), reports (REPORTFLEX), and SQL scripts (SQLFLEX). All of these packages with the exception of SQLFLEX, use a WYSIWYG approach to programming: that is, menus, screens, and/or reports are defined by painting their image in an editor of your choice. The coding syntax for these packages is consistent to facilitate learning.

Infoflex screens and reports are so versatile that much, if not all, of the features of an application can be defined without the use of a procedural language. This is true for such features as multi-record scrolling/updating, report query screens, menu chaining, and online field-specific help.

Though not a requirement of the Infoflex development process, C language routines can be easily linked with Infoflex screens, reports, and/or menus rendering no application too complex or unusual for Infoflex implementation. Developing C functions is made easy and powerful through our 4GL-like library and direct access to Infoflex variables.

Purpose of this User's Guide

The purpose of the Infoflex User's Guide is to describe how a simple application, such as the Infoflex Demo, is developed.

In this guide you will learn the basic features of the three types of forms: screen, report, and menu.

All features of Infoflex are fully explained in the Infoflex Reference Manual.

A complete source code listing of the demo software is contained in the Appendix of this User's Guide.

The Relational Database

Infoflex is designed to access a relational database which is a collection of related tables or files of information. Each table is comprised of any number of records of information consisting of one or more fields.

A simple example of a relational database is a master/detail relationship as in an invoice. The master information is stored in one table that contains data relating to the overall sale, such as customer and sale date. The detail information is stored in a different table that contains information about the description of each item being sold, its cost, and quantity. These two tables are then related based on their common information such as the invoice number.

Infoflex uses the same database structure as Informix database. This means that you can have an Infoflex or Informix program manipulate the same database.

Also, both Infoflex **SQLFLEX** and Informix SQL use the same syntax for database creation.

Comparison With Informix

For those familiar with Informix 4GL, Infoflex is a comparable product in developing large, sophisticated relational database applications. Infoflex, as well as being compatible with Informix's standard database, also uses a similar syntax to Informix's Perform package. Infoflex uses the Perform syntax to create menus, reports, and screens. As a result, Infoflex's interface is easier to learn and will reduce development time.

Another advantage of Infoflex over Informix is that the programs are smaller and program start-up is much faster. For example, the Demo program is driven by a single 330K executable program. This executable is loaded once and has the potential of driving any number of reports, screens, and menus.

The Audience

References to *you*, in this manual refer to the Infoflex developer. References to the *user* refer to the person that uses the application that *you* developed.

Documentation Organization

This manual is divided into chapters which are divided into topics. Chapter titles are in boldface and centered; Topics titles are in boldface and flush left like the topic title above this paragragh. For example, this topic is **Documentation Organization** within the chapter, **Introduction to Infoflex**.

2. BUILDING THE DATABASE

Basic Function

The first step in developing an application is to create collection of related tables or in other words, a database.

The SQL command file

SQLFLEX is the Infoflex package that will allow you to build, modify and query database tables. To perform this function, **SQLFLEX** reads commands from a command file created by you. The **SQLFLEX** command file is executed as follows:

fxsql *commandfile*

Creating the Database

The **SQLFLEX** command file to create the storage area for the demo database is as follows:

```
create database demo;
```

Assuming you place this statment in the file **demo.sql**,

fxsql demo

will create the database. The **demo** database will be created in the current directory under the directory name **demo.dbs**. The environment variable **FXDATA** can be used to overide where **fxsql** will look for the database.

This step must be done before creating any tables.

Creating Tables

The **create table** statement will define the name of the table to be created as well as define the field names and their data types. The **create index** statement is another appropriate statement of this command file and defines an index for a table.

The **tbagent** table of our demo is created with this command file:

```
create table tbagent (
    code          char(4),   /* this is a comment */
    lname         char(15),
    fname         char(10),
    hire_date     date,
    socno         char(13),
    raise_date    date,
    paymethod     integer,
    salary        money,
    commission    float
    );

create unique index tbagentkey
    on tbagent   (code);

create unique index tbagentkey2
    on tbagent   (lname, code);
```

Chapter 8, **Database Data Types**, in the Reference Manual describes the use of all the data types of Infoflex.

Assuming you place these statements in the file **tbagent.sql**,

fxsql tbagent

will create the table in your database.

SQLFLEX command files are portable to Informix SQL and furthermore, databases created by **SQLFLEX** are compatible with Informix.

The SQLFLEX command file listings for creating the demo database tables are in Topic 2 of the Appendix.

3. A SIMPLE SCREEN FORM

Basic Function

A screen form is that part of an application that allows a user to interactively add or modify information in a database or look up information in a database based on some index.

SCREENFLEX is that part of the Infoflex development environment that enables you to build the screen forms of an application.

The simplest screen form will display and allow modification to the fields of a single record of a single table.

The complete source code listing for the simple screen we are about to discuss is in Topic 3 of the Appendix. This is an agent table maintenance screen. When running the demo, the simple screen is invoked with option 3 from the main menu.

The Screen File

The Infoflex programmer defines how a screen is to look, the table or tables it accesses, the fields that are displayed, and the attributes of the fields. Using a text editor, you create the screen definition in a file with a **.flx** file name extension.

The four sections that comprise the screen definition file are: **TABLES**, **SELECT**, **SCREEN**, and **ATTRIBUTES**.

The TABLES Section

The **TABLES** section is where you will list all the tables to be accessed by a screen form and should be the first section in the screen definition file. Our demo simple screen accesses an agent table called **tbagent**. Here is the **TABLES** section:

```
TABLES
    tbagent
END
```

Each table defined in this section should have been previously created using **SQLFLEX** or Informix SQL.

The SELECT Section

The **SELECT** section defines the manner in which data is retrieved from the database, in this case from the **tbagent** table. The **SELECT** section for our agent screen looks like this:

```
SELECT
    tbagent( code)
        EXTRACTALL
END
```

The **code** argument is the field of **tbagent** by which the selected records will be ordered. This field or fields therefore must comprise an index of the associated table. **EXTRACTALL** indicates that all qualifying records will be selected. The user will then page one record at a time with the screen form.

The SCREEN Section

The screen layout is defined in the SCREEN section. Here is a compressed version of the demo screen:

```
SCREEN  scemp
{
        Agent Update Screen          DATE:[tday     ]



  Code <[eno ]>                   Last Raise:      [erdate  ]

  Last Name:  [elname          ]  Pay Method:      [e]

  First Name: [efname     ]       Commission Rate:[ecom]%

  Soc Sec No.:[esocno        ]    Salary :        $[esalary ]

  Date Hired: [ehdate  ]



}
END
```

Follow the **SCREEN** keyword with the screen name. Bracket the lines of the screen layout with { }. Bracket with [ ] the fields where data is input or displayed. Give each field a unique name or tag within the [ ]. In the following **ATTRIBUTES** section, we may or may not map the field tags to database table fields.

The < > around the **eno** field is not required and is just a convention we use to denote a key field of the screen. For more on this key field read the topic, **Running the Screen Form**.

The ATTRIBUTES Section

This section describes how each field of the form is used. List the field tags in the order that the screen cursor is to move through them. The form of each field definition in the **ATTRIBUTES** section is as follows:

```
ATTRIBUTES
     fieldtag  =  table . field ,  attributes1 ,
                  .  .  .  attributeN ;
                          .
                          .
                          .
END
```

The *table.field* is the database table field that corresponds to the screen field. In the demo, we defined the following screen field:

```
elname = tbagent.lname, ...
```

When the user retrieves a **tbagent** record with this screen, the data of the **lname** field of the **tbagent** table will automatically display in the **elname** screen field. When he saves input for this screen field, the input is saved in the **lname** field of the **tbagent** table record. When a screen field is associated with a table field, the screen field automatically inherits the data type of the table field.

The *table.field* parameter may also be replaced by a non-table or display-only designation:

```
tday = displayonly type date, ...
```

The **tday** tag has no associated database table field. You must explicitly state the data type of a **displayonly** field. See Chapter 8, **Database Data Types**, of the Reference Manual.

Screen fields may have an assortment of attributes that can be listed in any order. The simple attributes taken up here deal mainly with the appearance and input to screen fields. More sophisticated attributes, which will be taken up in the next chapter, allow you to do table look-ups, bring up help screens, etc. For now, let's look at some simple examples in the agent

screen demo.

There are several ways to display information to a screen field. One method is the automatic display of table field information; another is with the **default** attribute as shown below.

```
tday = displayonly type date, ..., default = today;
```

This use of the **default** attribute will cause today's date to display to the tday field. **Today** is a special keyword that represents today's date when used in place of a quoted date string expression in the form of "*mm*/*dd*/*yy*".

Here are the complete attributes for our first field:

```
tday = displayonly type date, noentry, noupdate,
       default = today;
```

It is not enough to specify a field as **displayonly** if you do not wish the user to address it. The **noentry** attribute prevents input to the field in **ADD** mode. **Noupdate** prevents input to the field in **CHANGE** mode.

Four other common attributes of input fields are **upshift**, **required**, **include**, **format**, and **comments**.

The **upshift** attribute will take lowercase alphabetic input and immediately convert it to uppercase.

In **ADD** or **CHANGE** mode the **required** attribute will prevent the user from leaving a field or form without filling in the required fields.

You may specify the range of acceptable values for a field with the **include** attribute. Any other user input will be rejected, and the user required to re-enter the field. In the demo we have as follows:

```
e = tbagent.paymethod, include (1 to 4, 8);
```

The values list in parenthesis can include individual values as well as ranges. An example of a valid expressions is: include(1 to 4, 8, 11 to 15, 20). Also, you need not quote values for string fields. The following expression is acceptable for a string field: include(ADAM to BOB, JOE). In this example, all names alphabetically between ADAM and BOB will be accepted. So BILL is legal input; CARL is not.

With the **format** attribute you may specify the format of numeric displays. There are two formatted numeric fields in the demo.

```
ecom =    tbagent.commission, format = "###.#";

esalary = tbagent.salary, format = "#,###.##";
```

The **comments** attribute allows you to define a message that displays at the bottom of the screen in row 22, when the screen cursor addresses a field. These are the attributes for the **elname** field:

```
elname = tbagent.lname, upshift,
         comments = "ENTER THIS EMPLOYEE'S LAST NAME";
```

Compiling the Screen Form

Once you have created the screen definition source file, you are ready to compile it into a file that can be run directly. The Infoflex utility **fxpp** will convert the screen source file into a **.pic** file. Given that the demo screen source file is **tbagent.flx**,

```
fxpp tbagent
```

will generate **tbagent.pic**. If **fxpp** encounters an error in **tbagent.flx**, **fxpp** will report it and terminate.

Note that the environment variable **PATH** must include the **.../fx/bin** directory path for Infoflex commands to operate.

Running the Screen Form

Once the screen is compiled, the user runs it with:

```
flex  tbagent
```

The default behavior of Infoflex screen forms places the user in **CHANGE** mode with the cursor on the key field of the screen. The key field is the one you specified in the **SELECT** section. In our demo this is **tbagent.code**.

Nearly all operations in a screen form are executed with a function key. At any time while running a screen form a ruler appears at the bottom of the display with the operative function keys labelled. In **CHANGE** mode, while on the key field of the screen, the ruler looks like this:

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
SAVE	HELP	SRCH	ADD	----	QRY	PREV	NEXT	FRST	LAST	----	DEL	----	----	----	----

In **CHANGE** mode the user first selects a record that he may then modify. Several function keys provide selection capability. In the demo **FRST** will select and display the first record of the **tbagent** table ordered by the **code** field. **LAST** will select the last record. Entering some value in the key field and pressing **NEXT** will bring up the next record in ordered sequence by that key. For example, if AA and AC are consecutive values for the key, entering AA or AB and pressing **NEXT** will bring up the record for AC. **PREV** works similarly but selects previous records. The **SRCH** and **QRY** allow you to search for records based on field values. These two keys are discussed in the chapter **MORE ADVANCED SCREEN FEATURES**.

With a record displayed, the **SAVE** option will update the database record with any changes the user has made in the screen form.

DEL will delete the record. Since **DEL** is such a destructive action, it will prompt the user to confirm the deletion.

At any time while in **CHANGE** mode, the **ADD** key will switch the form to **ADD** mode where, after filling out the form, the user may insert a new record into the database.

When in **ADD** mode the active function keys are:

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
SAVE	HELP	----	CHG	----	----	----	----	----	----	----	----	----	----	----	----

CHG will toggle back to **CHANGE** mode.

The **HELP** function key will be discussed later.

At anytime **ESC** will abort and exit the screen (**ESC ESC** on UNIX or XENIX systems). If there are changes or additions that may be lost with this exit, users are first asked if they wish to save the record.

Flex tbagent will bring up the **tbagent** screen in **CHANGE** mode. Alternately a screen may come up in **ADD** or **SEARCH** mode. In these cases you must explicitly state on the **flex** command line the initial mode of the screen. At the same time, you must also specify the other modes available to the screen. The command:

```
flex  tbagent  asqdc
```

initially places the **tbagent** screen in **ADD** mode.

4. THE SCREEN ARRAY

What Is It?

A screen array allows the display of multiple database records in a screen form at the same time.

The complete source code listing for the screen we are about to discuss is in Topic 4 of the Appendix. This is a travel agency booking screen. When running the demo, the array screen is invoked with option 4 from the main menu.

Joining Tables

When there is an array in a screen form, the **SELECT** section will typically define the relationship between the header record and the array records. Here is the **SELECT** section of our second demo screen, **bkinput.flx**:

```
SELECT
    bkmaster( bkno)
        EXTRACTALL
    bkdetail( bkno)  bkmaster( bkno)
        EXTRACTALL
END
```

For each selected **bkmaster** record, **bkdetail** records are selected that have the same value in their **bkno** field as in the **bkmaster**'s **bkno** field. These two fields are joined, establishing the relationship between the two tables. It happens in this example that the join fields have the same name, but this would not have to be the case. In the join expression:

```
bkdetail( bkno)  bkmaster( bkno)
```

bkno must be an index of **bkdetail**, but the **bkno** of **bkmaster** would not have to be an index of bkmaster.

The final **EXTRACTALL** parameter specifies that all records of bkdetail are selected that qualify. Optionally, you may instead specify **EXTRACT**, which would only select the first qualifying record. Obviously **EXTRACT** is not appropriate for filling out a screen array.

The SCREEN Section

Here is an abbreviated version of the **bkinput SCREEN** section:

```
SCREEN  bkinput
{
      BOOKING ENTRY SCREEN       DATE:[today    ]

Booking #: <[bkno  ]> Name <[bclient             ]>

Agent:<[code]> [fname      ][lname          ]
_____________________________________________
Sup-   Ship   Depart                   Sale
plier  Name   Date      Description   Amount
_____________________________________________
[bsup  |bship |bdepart |bdesc           |salamount]



Supplier Name:[bsupname          ]
}
END
```

You need define only the first line of the array. The fields that make up the array are **bsup** through **salamount**. Note that the pipe character "|" character is an alternate delimiter marking the end of one field and the beginning of the next. It is used to save space for field definitions.

REPEAT Within ATTRIBUTES

The **REPEAT** block within the **ATTRIBUTES** section brackets the array field. The number of array lines is specified after the **REPEAT** keyword as shown below.

```
REPEAT(5)
     bsup    = bkdetail.supplier, ...
     bship   = bkdetail.ship, ...
     bdepart = bkdetail.departdate, ...
     bdesc   = bkdetail.descript, ...
     salamount= bkdetail.salamount, ...
ENDREPEAT
```

The screen array will display five lines of **bkdetail** records.

Executing a Screen Array

To execute the array program **bkinput.flx**, you will enter the following command.

```
flex  bkinput
```

When the screen appears, the cursor will address the key field **bkno** for you to select a **bkmaster** record. To move to the array portion of the screen press the **SAVE** key. This action causes the array of **bkdetail** records to be displayed and the cursor to be positioned on the first row of the array. The array will only contain **bkdetail** records that successfully join with the **bkmaster** record.

By default the initial mode in the array is **CHANGE** mode if records exist or **ADD** mode if they do not. The function key ruler for the screen array appears the same as for the screen header. In array screens, however, **PREV**, **NEXT**, **FRST**, and **LAST** behave differently than in the simple screen. **NEXT** will display the next page of records, where a page is the number of records that can fit on one screen. **PREV** will display the previous page of records. **FRST** will display the first page, **LAST** the last page. Up and down arrows move the cursor up and down rows of the array.

Pressing the **ADD** key or cursoring off the end of the array will open a new line at the end of the array and automatically invoke **ADD** mode.

The **DEL** key will delete the current row and pull up rows below to close the gap.

The database is updated when the user cursors off an array row or presses **SAVE**. The **SAVE** key, in addition to saving the record, will also clear the array display and return the cursor to the key field of the screen form header.

Pressing the **ESC** key from the array (**ESC ESC** on UNIX or XENIX systems) exits the entire screen in the same manner as from the screen header.

The Detail-Only Array Screen

It is possible to have a screen form that is merely an array of records, without a header section. An example would be to take the Simple Screen Form, redesign the screen, and put a **REPEAT/ENDREPEAT** block around all of the fields in the **ATTRIBUTES** section. For more on the detail-only array screen see Topic 3.3, SELECT SECTION, in the Reference Manual.

5. MORE ADVANCED SCREEN FEATURES

The Lookup Attribute

Very often the value or code entered in a field must be verified. This is accomplished in simple cases with the **include** attribute described earlier. Other cases may require the value exist in a table. This is accomplished with the **lookup** attribute. To use this attribute, the field being looked up must be indexed in the table.

Below is an example of how the **lookup** attribute is specified.

```
code = bkmaster.agent, lookup( tbagent.tbagentkey), ...
```

In this example the **code** field will be looked up in the **tbagent** table using the **tbagentkey** index. The user will not be allowed to enter a **code** value that is not found in the **tbagent** table.

Lookup allows additional parameters for assigning fields from the lookup table to the screen form. An example of this is as follows.

```
code     = bkmaster.agent,
           lookup( tbagent.tbagentkey,
                   bkinput.lname = tbagent.lname,
                   bkinput.fname = tbagent.fname), ...
fname = displayonly type char, noupdate, noentry;
lname = displayonly type char, noupdate, noentry;
```

The values of the table fields **tbagent.fname** and **tbagent.lname** will be displayed in the screen fields **bkinput.fname** and **bkinput.lname**. This is a common usage of the **lookup** attribute, for often the looked-up value is a code and needs a more descriptive field to go along with it.

The Helpselect Attribute

The **helpselect** attribute is used to provide a popup list of valid codes from another table. While on the popup list, the user can search and select any valid code. The **helpselect** is initiated by pressing the **HELP** function key.

An example of syntax for the **helpselect** attribute is as follows.

```
bsub = bkdetail.agent, helpselect(pointagent)
```

The *pointagent* is an Infoflex array-only screen that will list the **tbgent** table fields, **code** and **name**, ordered by the **tbagentkey** index. The Appendix source file bkinput.flx shows the *pointagent* screen.

In addition to the standard array-only features, the **helpselect** screen provides 2 special methods for locating records: cursor sorting and character postioning.

Cursor sorting means the rows will be sorted based on where the cursor is positioned. If the cursor is positioned on the *Agent Code* field, the rows will be sorted by *Agent Code*. Likewise, if the cursor is positioned on the *Agent Name* field the screen will be sorted by the *Agent Name* field. To cursor from field to field on a Help screen, you must use the **TAB** key. Note that the field must have the **searchby** attribute for it to be sorted.

The character positioning feature allows you to type characters to locate records in the Help screen. Each character you press will reposition the screen to the closest match. To restart the character positioning (throw away previously entered characters and start over) press the **UP** or **DOWN** arrow keys. You may also press the **TAB** key to perform character positioning on a different field.

Once you have located the desired code on the Help screen, you may transfer the code to the original screen by pressing the

SAVE or **ENTER** key. You will then be returned to the original field with the selected code assigned.

Pressing **ESCAPE** will exit without effecting the original screen.

Online Help

Infoflex provides field specific help when pressing the **HELP** key. While on the help screen, you can press the **JUMP** key to access other levels of help. See Chapter 11, THE HELP SYSTEM, of the Reference Manual for the details on setting up a help system for your application.

Point and Shoot Screen

We have seen above how **helpselect** provides a point and shoot screen for selecting codes from a table. This section shows how to program your own point and shoot popup screen. To implement the point and shoot screen you will need to define the point and shoot screen layout, specify where and how the screen may be called, and develop the userexit function that will call the screen.

The following is the screen layout for the point and shoot screen used in the Demo program (see Appendix source for bkinput.flx). Note that this screen is a normal array-only screen.

```
SELECT
   tbagent( lname, code)
      EXTRACTALL
END

SCREEN  pointagent box popup reversebar window( 10, 18)
{
 Agent  Last            First
  [code|lname          |fname     ]




 Press SAVE to Select or ESCAPE to exit
}
END

ATTRIBUTES

REPEAT(6)
code         = tbagent.code, upshift,
               searchby( tbagent.tbagentkey);
lname        = tbagent.lname, noupdate, noentry,
               searchby( tbagent.tbagent2key);
fname        = tbagent.fname, noupdate, noentry;
ENDREPEAT

END
```

To specify where and how the screen will be accessed, any one of the userexit functions may be used. The demo uses the **helpkey** userexit so that the screen will be initiated upon pressing the **HELP** function key. The Point and Shoot screen will only be called from those fields where the **helpkey** is placed. Below is an example of its use.

```
   code = bkmaster.agent, helpkey( pointshoot);
```

The **helpkey** userexit specifies the "C" function that will be called when the user presses the **HELP** key. It is this "C" function that calls the point and shoot screen **pointagent**. This "C" function must be defined in the **INSTRUCTION** section of the program. The following is a listing of the "C" function used by the demo.

```
static pointshoot()
{
/*This userexit calls the Point and Shoot Selection Screen*/
 if (SAVEKEY == flexcmd("flex bkinput -f pointagent v-v")){
      /* move selected values to bkinput screen fields */
     move( @tbagent.code,  @bkinput.code);
     move( @tbagent.lname, @bkinput.lname);
       /* display effected bkinput screen fields */
     tmaprng( @bkinput.code, @bkinput.lname);
     }
 return(R_MREPAINT);/*causes bkinput screen to be repainted*/
}
```

The Notepad Screen

To implement Notepad screen you will need to define the Notepad screen layout, specify where and how the screen may be called, and develop the userexit function that will call the screen.

The following is the screen layout for the Notepad screen used in the Demo program. The Notepad screen is a normal array screen with one special keyword in the **ATTRIBUTE** section called **sequence**.

```
SELECT
   bkdetail( bkno, recno)
       EXTRACTALL
   bknote(bkno, recno, seqno) bkdetail( bkno, recno)
       EXTRACTALL
END

SCREEN  bkscrnote popup box window( 12, 15)
        auser1key(fxmovekey, MOVE)
{
         Notes       [bkno      ][recno   ]
[note                                      ]
[seqno  ]




}
END

ATTRIBUTES

bkno  = bkdetail.bkno, nodisplay, beforedit( be_bknote);
recno  = bkdetail.recno, nodisplay;

REPEAT(6)
seqno   = bknote.seqno, sequence,
          noupdate, noentry, nodisplay, setrow(-1);
note    = bknote.note;
ENDREPEAT

END
```

To specify where and how the screen will be accessed, any one of the userexit functions may be used. The demo uses the **auser1key** userexit so that the screen will be initiated upon pressing the **NOTE** function key. Below is an example of its use.

```
SCREEN  bkinput frame auser1key( auser1note, NOTE)
```

Using the **auser1key** also means the Notepad screen may be called by any of bkinput's array fields. The **auser1key** userexit specifies the "C" function that will be called when the user presses the **NOTE** key. It is this "C" function that calls the

Notepad screen **bkscrnote**. This "C" function must be defined in the **INSTRUCTION** section of the program. The following is a listing of the "C" function used by the demo.

```
static auser1note()
{
 /* This userexit calls the Notepad Screen (bknote)*/
 if (fxasave() < 0) /*save current array line record*/
     return(-1);
 flexcmd( "flex bkinput -f bkscrnote cs-cdsa");
 return(0);
}


static int be_bknote()
{
   /*
     This userexit is called upon entering the Notepad
     screen field bknote.bkno.
     Its purpose is to automatically fill the
     key fields (bkno and, recno) and then simulate the user
     entering the SAVE function key so that the user
     will immediately goto the note array.
     These key fields setup the join
     relationship between the bkinput line item and the
     notes for that line item.
   */

 move( @bkdetail.bkno, @bkscrnote.bkno);
 move( @bkdetail.recno, @bkscrnote.recno);
 flexkey = SAVEKEY;
 return(-1);
}
```

The *fxmovekey* userexit on the **bknote** screen is a standard Infoflex userexit that may be used with sequenced screens. This userexit provides a **MOVE** function key so the user can move rows on the array. To move rows, the user presses the **MOVE** key while on the source row. Next, the user cursors to the destination row and presses the **MOVE** key again.

Control Keys

There are a number of Control keys that work on all on screens. These Control keys perform very useful functions and are listed below.

CTL-D Saves the current screen values as defaults. These defaults will appear when in **ADD MODE** or on report selection screens. Each user can have his own defaults by setting the environment variable **FXDEFAULT** to a user-specific directory.

CTL-N Calls the Accounflex menu from wherever you are in the system. You will be returned to your current postion upon returning from the menu.

CTL-P Repeats the previously entered value.

CTL-T Prints the screen image to the default printer.

CTL-W Writes the screen image to disk. You will be prompted for a filename for storing the image. The filename you enter will be appended with the suffix '.scr'.

Search Mode

As an alternative to searching on the *key* field of a form, **SEARCH** mode provides the capability of record selection by other indices of the record.

SEARCH mode is turned on by specifying the **searchby** attribute for one or more screen fields. When this mode is active, the **F5** key will be labeled **SRCH** in **CHANGE** mode.

Let us look at an example in our demo. In the **ATTRIBUTE** section of the **bkinput** demo screen of the previous chapter we have:

```
bclient = bkmaster.name, searchby( bkmaster.bkmname)
```

Bkmaster.bkmname is the name of an index associated with the **bkmaster.name** field. Any fields that are themselves indices or parts of indices are candidates for a **searchby** attribute.

When the user presses the **SRCH** key, those fields with the **searchby** attribute will be underlined, and only those fields can be cursor addressed. Here the **PREV**, **NEXT**, **FRST**, and **LAST** keys are available and make their record selections based on the index specified in the **searchby** attribute.

Pressing the **EXIT** key will exit **SEARCH** mode and return to **CHANGE** mode with the selected record.

Query Mode

The Query feature, unlike the search feature, allows you to search on any field or combination of fields and use wildcard or relational operators.

Query is active when the function key label **QRY** is displayed. Upon pressing **QRY**, the screen fields that are queriable will be underlined and the **QUERY MODE** message will appear at the top of the screen. While in **QUERY MODE** you may query on any of the underlined fields by cursoring to the desired field then entering the value you wish to query on. Values may be entered for as many fields as you want.

The query values you enter may include special operator characters that provide enhanced searching capabilities. Below is a table of operators that may be included with the query value.

Operator	Operator Name	Compatable Data Types
=	Equal	all
>	Greater than	all
<	Less than	all
>=	Greator than or equal	all
<=	Less than or equal	all
<>	Not equal	all
|	OR	all
&	AND	all
*	Wildcard for any number of character	CHAR
?	Wildcard for 1 character	CHAR
:	Range	all

When using any of the first eight operators place the operator at the start of the query value.

Use the '=' operator only when you want to find NULL values in a character field. In this case you would just enter the '=' operator by itself.

WILDCARD OPERATORS

Wildcard operators (*, ?) can only be used in character fields. Querying with wildcard operators is best described with examples. For example, specifying the query value "*corp*" would find all records with the word "corp" anywhere in that field. The following list of values would match this query value.

1) corporation
2) IBM Corporation
3) Marine Corp

Note that the query is not case sensitive.

The query value "corp*" would only find records where the field starts with the value "corp". In this case only the first value in the above list "corporation" would match.

The wildcard operator (?) is a one character wildcard. For example, the query value "????corp*" would only match "IBM Corporation" on the above list.

RANGE OPERATOR

The range operator (:) is used to specify a range. It lets you search for all values that lie between one value and another. The range is inclusive.

For example, to search for all zip codes from 94010 and 95080, enter "94010:95080" as your query value. Query will find all records where the value of the field lies within the specified range.

OR and AND OPERATORS

Query assumes that all entered query values must match the record for it to be selected. The OR (|) operator allows you to select the record if either query values match. The OR (|) operator is placed at the beginning of each query value.

While in the **QUERY MODE**, the function key labels will appear as follows.

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
RUN	HELP	ORUN	CLR	----	EXIT	----	----	----	----	----	----	----	----	----	----

To start the query, press the **RUN** function key. After all of the records have been found, a message will appear at the bottom of the screen showing the number of matches found. You will then be returned to the original screen where you will be able to use the **NEXT, PREV, FRST, LAST** function keys to view the selected records.

When you return to the original screen, the mode message will be appear with asterisks ***CHANGE MODE*** letting you know you are looking at a query list.

To clear the query list, you must return to the **QUERY MODE**, clear all of the query values (press the **CLR** function key), and then rerun the query (press the **RUN** function key). When you return to the original screen you will be able to access all records.

6. A SIMPLE REPORT

Basic Function

Frequently a user may wish to look at many records of a table at once or make a hardcopy presentation of this information. The format of such output should be easily readable and meaningful to the user.

REPORTFLEX is that part of the Infoflex development environment that enables you to build the report forms of an application.

The complete source code listing for the simple report we are about to discuss is in Topic 5 of the Appendix. When running the demo, the simple report is invoked with option 5 from the main menu. The report shown in the demo is the agent summary report.

The Report File

The Infoflex report file has many things in common with the Infoflex screen file. Like the screen file, the report definition file name has a **.flx** extension.

The **TABLES** section of a report file is identical to that of the screen file. It comes first in the file and specifies all tables accessed by the report.

We will now describe the other sections which make up the report definition file.

The SCREEN Section

The **SCREEN** section and its accompanying **ATTRIBUTES** section in a report file are optional. Normally, you will include these sections to allow the user to choose the form of output and the number of copies. Here is a compressed version of the **SCREEN** section and its **ATTRIBUTES** section in the demo report file **tbagentr.flx**:

```
SCREEN  select
{
                 REPORT SELECTION SCREEN
                     Agent Code Table

 Report Destination: [d  ]  (S=Screen, P#=Printer,
                             D=Disk,   A=Aux)
 Report Copies:      [c ] (1 - 10)

}
END

ATTRIBUTES
    d(rptdest)   = displayonly type character,
                   required, upshift;
    c(rptcopies) = displayonly type smallint,
                   required, include( 1 to 10);
END
```

You must name the report screen **select**. The **rptdest** field tag is reserved for this special screen. Note that the tagname **"d"** is renamed to tagname **rptdest** in the ATTRIBUTE section. If the value input to the **rptdest** field is **"S"** the output of the report will display to the screen. An input of **"P"** will output the report to the system printer. Entering a **"D"** will output the report to a disk file for later access. **"A"** will direct the report to the auxiliary port of the terminal where such a port exists. The **rptcopies** field tag is reserved for the field that specifies the number of copies of the report. One to 10 copies of a report can be generated with a single request. Note that the attributes for the **rptdest** and **rptcopies** fields insure proper input to these fields.

Without this **SCREEN** and **ATTRIBUTES** section, one copy of the report will be displayed directly on the screen.

The SELECT Section

The next section of the report definition is the **SELECT** section. In our simple demo report, since we are selecting all the records of the **tbagent** table ordered by the **code** field, the **SELECT** section is identical to that of our simple demo screen:

```
SELECT
     tbagent( code)
         EXTRACTALL
END
```

The REPORT Section

The section immediately following the SELECT section is the REPORT section. The REPORT section is used layout each subsection of the report (headers, detail, totals, and footings). Our simple report has a header, then a body or the detail of the report, and finally a total. Here is how the REPORT section defines our simple demo report:

```
REPORT

heading
{
            AGENT TABLE REPORT             PAGE:[page]
                                    [tim     ][tday    ]

 Code   Last Name              First Name  Salary
 -----  -------------------- ----------- ----------
}

detail
{
 [eno ] [elname               ][efname    ][esalary ]
}

total
{
                                           -----------
                          Payroll Total:   [esalary  ]
}

END
```

In the report output, the **heading** subsection will output at the top of every page, and the **detail** subsection will output for each **tbagent** record read. The **total** subsection will be printed at the end of the report and will display the accumulated total of **esalary** for all **tbagent** records that have been selected for the report.

The ATTRIBUTE Section

The **ATTRIBUTE** section is where the field tags are defined for the **REPORT** section. This is done exactly like the **ATTRIBUTE** section for screens. Some field tags used primarily for reports are described below. **Page** is a reserved field tag that will cause the page number to be printed in that field. Since the **page** field is not associated with a table field, its attribute must be:

```
page = displayonly type smallint;
```

Typically reports are time and date stamped. You achieve this by assigning the keywords **today** and **time** to the default attributes of the **tday** and **tim** fields:

```
tday = displayonly type date, default = today;
tim  = displayonly type mtime, default = time;
```

The **mtime** field data type is the time of day in military format, the 24-hour clock.

The only other attributes that are meaningful in a report are **left**, **right** and **format**. The **left** and **right** attributes will left

and right justify a value within a field. **Format** will allow you to specify the format of a numeric value in the same way as the **format** attribute of the screen form.

Compiling and Running the Report

Compiling and running reports is the same as for screens. To compile your report, use the **fxpp** command as follows.

```
fxpp  tbagentr
```

The **fxpp** command will compile our **tbagentr.flx** demo report, generating the file **tbagentr.pic**.

The **flex** command is then used to run the compiled report.

```
flex  tbagentr
```

will execute the report.

If the report uses a **select** screen, pressing **RUN** from a properly filled out **select** screen will initiate the report process. If the report is directed to the screen, the first page of output is displayed with the following function key ruler:

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
EXIT	----	JUMP	----	SRCH	----	PREV	NEXT	FRST	LAST	PRNT	C132	----	SHFL	SHFR	----

These keys provide a variety of ways of moving through the report. **NEXT** pages forward through the report. **PREV** pages backwards. **FRST** displays the first page of the report. **LAST** displays the last page of the report. **JUMP** will prompt the user for a page number to display. **SRCH** will prompt for a character string pattern to search for in the report and if found will display the page.

You may press **SHFR** to right shift the display to view columns beyond 80. **SHFL** will shift the display back to the left. Some terminals will support character compression to 132 columns. The **C132** key will place such terminals in that mode.

Finally, the **PRNT** key will direct the report to the system printer.

The **EXIT** key will exit back to the **select** screen if one exists. Press **ESC** from the **select** screen (**ESC ESC** on UNIX or XENIX systems) to leave the report altogether.

7. MORE ADVANCED REPORT

What Are Its Features?

The advanced report example given in this chapter introduces a number of additional report features. These features include a **select** screen for letting the user specify which records of a table to report, table joining, break totals, and sorting.

The complete source code listing for the more advanced report we are about to discuss is in Topic 7 of the Appendix. Th report shown in the demo is a sales by agent report. When running the demo, this report is invoked with option 6 from the main menu.

Selection Criteria

Our first demo report printed all the records of the **tbagent** table. You may reference database fields with a **select** screen that will allow the user to limit the records selected. The following report **SCREEN** section and its **ATTRIBUTES** section are an abbreviated version of the demo report **saagent.flx**. This report selection screen allows the user to specify which agents to include in the Sales Report.

```
SCREEN  select
{
               REPORT SELECTION SCREEN
                    Sales by Agent


 Report Destination: [d  ]  (S=Screen, P#=Printer,
                             D=Disk,   A=Aux)
 Report Copies:      [c ]   (1 - 10)

          Agent:  1) [agen]
                  2) [age2]
                  3) [age3]


}
END

ATTRIBUTES

     d(rptdest)   = displayonly type character,
                    required, upshift;
     c(rptcopies) = displayonly type smallint,
                    required, include( 1 to 10);
     agen = bkmaster.agent, upshift;
     age2 = bkmaster.agent, upshift;
     age3 = bkmaster.agent, upshift;

END
```

Each of the three agent fields defined on the screen is tied to the database table field **bkmaster.agent**. On this particular **select** screen, the user may specify up to three specific agents to report. If the user specifies no agents, then all **bkmaster** records are selected.

If, instead of three agent field defined on the screen we had two, the report program would treat the two fields as a range. For example:

```
   Agent Range: [agefirst]  to  [agelast ]
```

As a side note, report **select** screen fields can have many of the attributes that screen forms have, including **lookup** and **helpselect**.

To enable the selection criteria feature in a report, a **WHERE** clause must be added to the **SELECT** section:

```
SELECT
    bkmaster( agent, bookdate)
        EXTRACTALL
    WHERE whereselect
END
```

Joining Tables

All relations between tables must be defined in the **SELECT** section. Here is the **SELECT** section of the demo report **saagent.flx**:

```
SELECT
    bkmaster( agent, bookdate)
        EXTRACTALL
    bkdetail( bkno)  bkmaster( bkno)
        EXTRACTALL
    tbven( code)  bkdetail( supplier)
        EXTRACT
    tbagent( code)  bkdetail( agent)
        EXTRACT
    WHERE whereselect
END
```

In the first join, records are selected from **bkdetail** by its **bkno** index. The value for **bkno** of **bkdetail** is determined by the value of **bkno** in a record selected from **bkmaster.**

Now we see the use of **EXTRACT**. It is known that there will only be one **tbven** record for a given supplier code, so there is no reason to look further.

Report Breaks

A report break may occur at any point in the selection process where the value of the sort index changes. Again, the sort index is established in the initial clause of the **SELECT** section:

```
bkmaster( agent, bookdate)
    EXTRACTALL
```

In this demo report we wish to print a new heading for each agent. The following heading definition accomplishes this for us. **REPORT** section of:

```
heading  breakon( bkmaster.agent)
{ ...
}
```

To print monetary totals for each week and each month of activity. You can break on different parts of a date. For example:

```
total  breakon( bkmaster.bookdate, 4)
```

will print a week's total. The specification:

```
total  breakon( bkmaster.bookdate, 2)
```

will print a month's total. 1 as the second parameter of **breakon** specifies a break on year; 3 a break on day.

Sorting By Any Criteria

Reports have the ability to order their information in more ways than supported by the single index of a master table. Using a temporary table, you can sort the records of a report by any of its fields.

The **ONINDEX** clause in the **SELECT** section creates a temporary table and defines the index for the table and thus the sort order of the report. The following example is the **SELECT** section of the demo report **saagent2.flx**. The complete source listing for **saagent2.flx** is in Topic 9 of the Appendix.

```
SELECT

    bkmaster( bookdate, bkno)
              bookdate
              agent
        EXTRACTALL

    bkdetail( bkno)  bkmaster( bkno)
              salamount
        EXTRACTALL

    tbagent( code)  bkmaster( agent)
              name
        EXTRACT

    WHERE whereselect
    ONINDEX bktemp( agent, bookdate)

END
```

The temporary file created is **bktemp**. The sort index is made up of the **agent** and **bookdate** fields from the **bkmaster** table. Note how the fields for the temporary table are specified. The **bookdate** and the **agent** fields from the **bkmaster** table, the **salamount** field from the **bkdetail** table, and the **name** field from the **tbagent** table make up the fields of the **bktemp** table. You may define the **bktemp** index to be any one or any combination of these fields.

8. BUILDING MENUS

Once you have developed all the screens and reports of your application, the final step is to organize them into menus.

MENUFLEX provides the tools that allow you to design, build, and compile menus for your application.

To build menus, select the **Design Menu** option on the **Development Menu**. An Infoflex screen will appear for creating and modifying menus. The **Menuflex** chapter in the **Reference Manual** describes the process of developing menus.

APPENDIX

1. Running the Demo

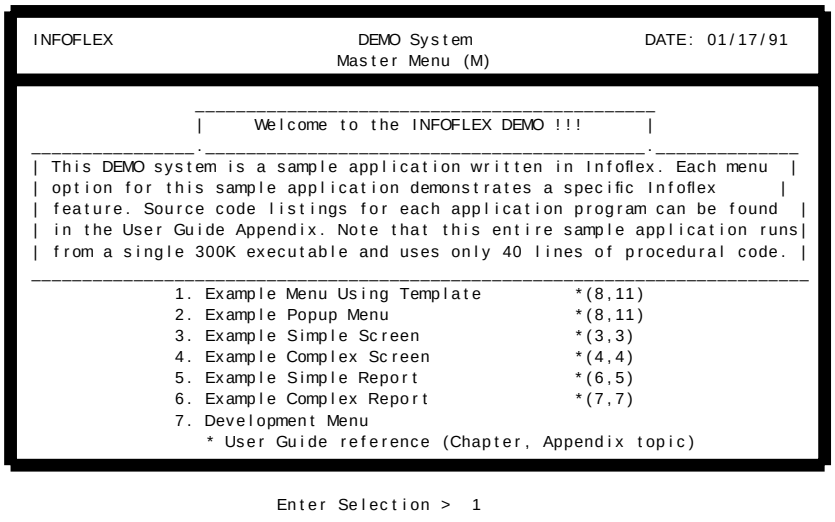
This topic describes how to run the Infoflex demo application. The installation of the demo will depend on your machine, operating system, and the media on which the demo is distributed to you. A precise set of installation instructions will accompany the demo.

Once installed, simply typing

demo

will execute the demo.

Here is the main menu of the demo:



While on this menu, you may select an option by positioning the cursor and pressing **RETURN** or by entering an option number and pressing **RETURN**. You may exit any menu choice, screen, and/or report by pressing the **ESCAPE** key once.

For your convenience each option is tagged with a Chapter number and an Appendix topic number as follows.

5. Example Simple Report (6,5).

This tag indicates that the simple report is discussed in Chapter 6 and the source code listing is in Topic 5 of the Appendix.

The following section lists the features you will see when you explore each main menu option.

1. Example Menu Using a Template. This option will enter a submenu, the design of which is specified by the **bookmenu** template in Topic 11 of the Appendix. The options of this submenu do nothing, but the submenu demonstrates the flow from one menu to another. Press the **ESCAPE** key to return to the main menu.

While on this menu you can try out the Menu Chaining feature. This feature will allow you to jump directly to any menu option in the demo system. To activate this feature, press the F3 key. The following prompt will appear at the bottom of the screen:

Enter Menu Chain word ===>

When this prompt appears type the menu code **"M"** and press **RETURN**. You will then go directly to the menu whose menu code is "M". In the case of the demo, the menu you will go to is the Master menu. Had we typed a menu option

number after the menu code (M5), that option would have been executed. For your convenience the menu codes are displayed within parenthesis after the menu title.

2. Example Pop-Up Menu. This option will display a bordered overlay or pop-up submenu. This particular menu has no template, and therefore its appearance is generated automatically. Press the **ESCAPE** key to return to the main menu.

3. Example Simple Screen. This option invokes the agent entry screen to demonstrate the basic screen form operations on a single database table record.

At the very bottom of the simple screen are listed the active function keys. This list varies depending on which field the cursor is on. To position to an existing agent record you may enter an agent code and press **RETURN** or press the **NEXT** function key (F8). You may also enter part of an agent code and press the **NEXT** key to move to the closest matching agent code.

To add an agent, press the **ADD** function key (F4). When you are finished, press the **ESCAPE** key to return to the main menu.

4. Example Complex Screen. This complex entry screen will demonstrate a number of features including: online text help, point and shoot table selection, popup data entry screens, master detail relation, multi-record scrolling/updating, tablehelp facility, and popup notepad entry.

When the complex entry screen appears press the **NEXT** function key to position to the first master record. This demo application screen is for entering trip bookings and will sometimes be refered to as the booking screen.

ONLINE HELP

While positioned on the first field of the screen, you can press the **HELP** function key to obtain information about the field you are on. This is the online text help system that is available for all fields. Once in the online help subsystem there are many levels of help that may be accessed by pressing the **HELP**, **NEXT**, or **PREV** functions keys. To exit online help press the **RETURN** or **ESCAPE** key and you will be returned to the booking screen.

HELP SELECTION

Next, you will want to see the help point and shoot selection screen. This popup screen lists all of the possible codes and allows you to choose one. To access the help selection screen, cursor to the agent field then press the **HELP** key. Upon pressing the **HELP** key, a list of agent codes and names are displayed. If you had entered an agent code prior to accessing the point and shoot screen, the point and shoot list will begin with the closest match. To select an agent, position the cursor onto the desired agent and then press the **SAVE** or **ENTER** key. You will then be returned to the booking screen with the agent code you selected. To return to the booking screen without making a selection, press the **ESCAPE** key.

While on the help selection screen, you may search for an agent code by typing characters. As you type characters the list repositions to the closest match. If you want to search by agent name, cursor over to the agent name field by pressing the **TAB** key. When you are cursored on the agent name field, the screen sort order changes from code to name. Also, as you enter characters the screen searches by name instead of code.

You may also search for agents using the **QRY** key. The **QRY** key puts you into **QUERY MODE** where you can do wildcard searches on one or more fields. Refer to the chapter **MORE ADVANCED SCREEN FEATURES** for information about **QUERY MODE**.

POPUP DATA ENTRY SCREEN

In addition to selecting agents, you may also enter new agents. This is accompished by pressing the **ZOOM** function key (F13 or shift F3) while on the agent code field. Upon pressing the **ZOOM** key a popup screen for entering agent information will appear. This screen behaves exactly like the simple screen discussed above. When you have finished entering a new agent, press the **SAVE** key or the **ESCAPE** key to return to the booking screen.

MASTER-DETAIL RELATION

Another important feature of the booking screen is the joining of an array of detail records to the master booking record. To join the array of detail records, press the **SAVE** function key (F1). This will move you to the lower portion of the screen where the detail records appear. While on the detail portion of the screen, you will want to play with

various function keys for scrolling pages (see labels at bottom of screen).

POPUP DATA ENTRY SCREEN (more)

The supplier field, like the agent field, also has a popup entry screen for entering new suppliers. To see this popup entry screen, position the cursor on the Supplier Code field and then press the **ZOOM** function key (F13 or shift F3). While on the popup data entry screen you may ADD, DELETE, or CHANGE information stored in the table. Pressing the **SAVE** key will save your entry and return you to the array screen. Pressing the **ESCAPE** key will abort the entry and return you to the array screen.

POPUP NOTEPAD SCREEN

To see the notepad data entry screen, press the function key labeled **NOTE** (F14 or shift F4). While on the note entry screen you may ADD, DELETE, or CHANGE notes for the detail record you were on. The notepad screen is essentially another multi-record scrolling screen joined to the booking detail record. Another feature worth noting is the **MOVE** function key. This key allows you to move lines by pressing the **MOVE** key twice; once on the source row and then next on the destination row. Pressing the **SAVE** or **ESCAPE** key will return you to the booking detail record.

To get back to the master portion of the booking screen, you can press the **SAVE** key. Pressing the **ESCAPEKEY** will return you to the menu.

5. Example Simple Report. This report simply lists all the data in a single database table. The initial screen you encounter will prompt for Report Parameters. To see the report, press the **RUN** function key.

6. Example Complex Report. The report joins data from several separate tables. It has a selection screen which allows you to vary the criteria by which records are selected. To see the report, press the **RUN** function key.

7. Development Menu. The Development Menu is a menu that comes with the Infoflex Development Package. The choices on this menu are described in the **Reference Manual**.

2. The Schema

```
create table menuhead (
    menuname      char(10),
    title1        char(76),
    title2        char(76),
    prevmenu      char(10),
    template      char(18),
    ncolumns      smallint,
    nmenufields   smallint,
    startcr       char(1),
    urow          smallint,
    ucol          smallint,
    lrow          smallint,
    lcol          smallint
    );
create unique index mhmenukey
    on menuhead ( menuname);

create table menufield (
    menuname      char(10),
    itemnum       smallint,
    itemdesc      char(60),
    exectype      char(1),
    execline      char(80),
    password      char(8),
    clearscr      char( 1),
    startcr       char(1),
    endcr         char(1)
    );
create unique index mfmenukey
    on menufield ( menuname, itemnum);

create table sysfile (
    sysname        char(10),
    company        char(40)
    );
create unique index sysfilekey
    on sysfile   (sysname);

create table bkmaster (
    bkno          serial(50),
    name          char(20),
    bookdate      date,
    agent         char(4),
    salamount     money
    );
create unique index bkmbkno
    on bkmaster (bkno);
create unique index bkmname
    on bkmaster (name, bkno);
create index bkmagent
    on bkmaster (agent, bookdate, bkno);
create index bkmbookdate
    on bkmaster (bookdate, bkno);

create table bkdetail (
    recno         serial,
    bkno          integer,
    bookdate      date,
    supplier      char(6),
    ship          char(6),
    departdate    date,
    descript      char(20),
    salamount     money
    );
create unique index bkdbkno
    on bkdetail   (bkno, recno);
create index bkdtrip
    on bkdetail   (supplier, ship, departdate);
create index bkdsupdepart
```

```
    on bkdetail   (supplier, departdate);
create index bkdsupbook
    on bkdetail   (supplier, bookdate);

create table bknote
       (bkno          integer,
        recno         integer,
        seqno         integer,
        note          character(45)
    );

 create unique index bknotekey
      on bknote (bkno, recno, seqno);

create table tbagent (
    code          char(4),
    lname         char(15),
    fname         char(10),
    hire_date     date,
    socno         char(13),
    raise_date    date,
    paymethod     integer,
    salary        money,
    commission    float
    );
create unique index tbagentkey
    on tbagent   (code);

create table tbven
    (code          char(6),
     name          char(15),
     contact       char(15),
     phone         char(15),
     saleflag     char(1),
     type          char(1)
     );

create unique index tbvenkey
    on tbven   (code);
```

3. The Simple Screen - tbagent.flx

```
TABLES
  tbagent
END


SELECT
  tbagent( code)
  EXTRACTALL
END


SCREEN  tbagent frame defaulton
{

 DEMO       [modemsg      ]      Agent Entry Screen             DATE:[tday    ]


        Code <[eno ]>                       Last Raise:     [erdate  ]

        Last Name:   [elname          ]     Pay Method:     [e]

        First Name:  [efname    ]           Commission Rate:[ecom]%

        Soc Sec No.: [esocno       ]        Salary :       $[esalary ]

        Date Hired:  [ehdate  ]

}
END


ATTRIBUTES

modemsg = displayonly type character, noupdate, noentry, reverse, retain;
tday = displayonly type date, noupdate, noentry, default = today, retain;
eno = tbagent.code, upshift, required, searchby(tbagent.tbagentkey), comments =
"Enter the agent's code number to identify this agent throughout the system";
elname= tbagent.lname,  searchby(tbagent.tbagent2key),
        comments = "Enter this employee's last name";
efname= tbagent.fname, comments = "Enter this employee's first name";
esocno = tbagent.socno, comments =
"Enter this employee's social security number";
ehdate = tbagent.hire_date,
         comments = "Enter the hire date for this employee";
erdate = tbagent.raise_date, comments =
"Enter the date of the last pay raise for this employee";
e(epaymeth)= tbagent.paymethod, include(1 to 4, 8), comments =
"Enter the method number (1-4, 8) by which this employee will be paid";
ecom =   tbagent.commission, format="###.#", comments =
"Enter the commission rate for this employee";
esalary = tbagent.salary, format="#,###.##", comments =
"Enter the yearly salary for this employee";

END
```

4. The Array Screen - bkinput.flx

```
TABLES
  menufield
  tbagent
  tbven
  bkmaster
  bkdetail
  bknote
END


SELECT

     bkmaster( bkno)
       EXTRACTALL
     bkdetail ( bkno) bkmaster( bkno)
       EXTRACTALL

END


SCREEN  bkinput frame   defaulton
                        zoomscreen( "tbvenpop")  azoomscreen( "tbagentpop")
                        auser1key( auser1note, NOTE)
{

 DEMO [modemsg      ]  BOOKING ENTRY SCREEN  [tday                  ][time    ]


  Booking #:<[bkno  ]> Name<[bclient             ]> Booking Date:<[bkdate  ]>

  Agent:<[code]>[fname     ][lname          ]

 @_____________________________________________________________________________
     Supplier              Ship    Depart                                 Sale
   Code   Name             Name    Date      Description                 Amount
 @_____________________________________________________________________________
  [bsup  |bsupname       ][bship ][bdepart ][bdesc               ][salam     ]




 @_____________________________________________________________________________
  Supplier Contact:[contact        ]Phone:[phone          ]Totals:$[saltot   ]
}
END

ATTRIBUTES

modemsg = displayonly type character, noupdate, noentry, reverse, retain;
tday = displayonly type date, default=today, noupdate, noentry, retain,
       format="Mmm dd yyyy Ddddddd";
time = displayonly type time, default=time, noupdate, noentry, retain;

bkno = bkmaster.bkno, searchby(bkmaster.bkmbkno), right,
       comments="Enter the Booking Number.";
bclient = bkmaster.name,upshift, searchby( bkmaster.bkmname), required,
   comments = "Enter the Client's Name";
bkdate= bkmaster.bookdate, searchby( bkmaster.bkmbookdate),
       default=today, required,
       comments = "Enter the Booking Date";
code = bkmaster.agent, upshift,
    searchby( bkmaster.bkmagent), zoomscreen( "tbagentpop"),
    lookup(tbagent.tbagentkey, bkinput.lname = tbagent.lname,
                          bkinput.fname = tbagent.fname),
    autohelp,
    helpselect(pointagent)
    comments="Enter Agent code (press the HELP key to select choice)";
fname = displayonly type char, noupdate, noentry;
lname = displayonly type char, noupdate, noentry;

REPEAT(6)

bsup = bkdetail.supplier, required, upshift,  zoomscreen( "tbvenpop") ,
    lookup(tbven.tbvenkey, bkinput.bsupname = tbven.name,
                          bkinput.contact  = tbven.contact,
                          bkinput.phone    = tbven.phone),
    tablehelp("demoflex tbven asdc", tbven.tbvenkey, tbven.code, tbven.name),
    searchby( bkdetail.bkdsupdepart),
comments = "Enter the supplier (press HELP to see a list of valid codes)";
bsupname = displayonly type char, noupdate, noentry;

bship = bkdetail.ship, upshift, left,
        comments = "Enter the ship name for Cruises";
bdepart = bkdetail.departdate, searchby( bkdetail.bkdsupdepart),
```

```
        comments = "Enter the departure date";
bdesc = bkdetail.descript,
        comments = "Enter any description you would like";
salam = bkdetail.salamount, total( bkinput.saltot), format="#,###.##"
        comments = "Enter total sale amount";

ENDREPEAT
contact = displayonly type character, noupdate, noentry;
phone   = displayonly type character, noupdate, noentry;
saltot = bkmaster.salamount, format="b#,###.##", noupdate, noentry;

END



TABLES
  tbagent
END


SELECT
  tbagent( code)
  EXTRACTALL
END


SCREEN  tbagentpop box popup window( 7, 35, 0, 0)
{

     AGENT UPDATE SCREEN
     @_____________________________
     Code <[eno ]>
     Last Name:   [elname         ]
     First Name:  [efname    ]
     Soc Sec No.: [esocno       ]
     Date Hired:  [ehdate  ]
     Last Raise:     [erdate  ]
     Pay Method:     [e]
     Commission Rate:[ecom]%
     Salary :       $[esalary ]
}
END


ATTRIBUTES

eno = tbagent.code, upshift, required, comments =
"Enter the agent's code number to identify this agent throughout the system";
elname= tbagent.lname, comments = "Enter this employee's last name";
efname= tbagent.fname, comments = "Enter this employee's first name";
esocno = tbagent.socno, comments =
"Enter this employee's social security number";
ehdate = tbagent.hire_date, comments = "Enter the hire date for this employee";
erdate = tbagent.raise_date, comments =
"Enter the date of the last pay raise for this employee";
e(epaymeth)= tbagent.paymethod, include(1 to 4, 8), comments =
"Enter the method number (1-4, 8) by which this employee will be paid";
ecom =   tbagent.commission, format="###.#", comments =
"Enter the commission rate for this employee";
esalary = tbagent.salary, format="#,###.##", comments =
"Enter the yearly salary for this employee";

END


TABLES
  tbven
END


SELECT
   tbven(code)
   EXTRACTALL
END


SCREEN  tbvenpop frame popup window( 7, 30, 0, 0)
{

 SUPPLIER UPDATE SCREEN

 Code <[vno   ]>
 Name  [vname               ]
 Type  [t]
 Sale Flag [s]
}
END


ATTRIBUTES
```

```
vno = tbven.code, upshift, required, comments="Enter THE SUPPLIER's CODE";
vname = tbven.name, comments = "Enter SUPPLIER's NAME";
t = tbven.type, upshift, required, include( T, C, O),
    comments = "Enter SUPPLIER TYPE: T= Tour, C=Cruise, O=Other";
s = tbven.saleflag, upshift, required, include( Y, N),
    comments = "SALE included in Sales Analysis Reports: Y=Yes, N=No";

END


SELECT
   tbagent( code)
      EXTRACTALL
END

SCREEN  pointagent box popup reversebar window( 10, 18)
        azoomscreen("demoflex tbagent")
{
 Agent  Last            First
  [code|lname          |fname     ]




 Press SAVE to Select or ESCAPE to exit
}
END


ATTRIBUTES

REPEAT(6)
code         = tbagent.code, upshift,
               searchby( tbagent.tbagentkey);
lname        = tbagent.lname, noupdate, noentry,
               searchby( tbagent.tbagent2key);
fname        = tbagent.fname, noupdate, noentry;
ENDREPEAT

END


SELECT
   bkdetail( bkno, recno)
       EXTRACTALL
   bknote(bkno, recno, seqno) bkdetail( bkno, recno)
       EXTRACTALL
END

SCREEN  bkscrnote popup box window( 12, 15)
        auser1key(fxmovekey, MOVE)
{
         Notes       [bkno      ][recno   ]
[note                                      ]
[seqno  ]




}
END

ATTRIBUTES

bkno  = bkdetail.bkno, nodisplay, beforedit( be_bknote);
recno  = bkdetail.recno, nodisplay;

REPEAT(6)
seqno   = bknote.seqno, sequence, noupdate, noentry, nodisplay, setrow(-1);
note    = bknote.note, savecol,
     comments="Enter Notes (press the SAVE or ESC key when done)";

ENDREPEAT


END



INSTRUCTIONS

static auser1note()
{
    /* This userexit calls the Notepad Screen (bknote) */
    if (isempty( @bkinput.bsup) == YES) {
       msggerr( "Supplier Code required");
       return(0);
       }
```

```
    if (fxasave() < 0)  /* save current array line record */
        return(-1);
    flexcmd( "flex bkinput -f bkscrnote cs-cdsa");
    return(0);
}


static int be_bknote()
{
     /*
        This userexit is called upon entering the Notepad
        screen field bknote.bkno.
        Its purpose is to automatically fill the
        key fields (bkno and, recno) and then simulate the user
        entering the SAVE function key so that the user
        will immediately goto the note array.
        These key fields setup the join
        relationship between the bkinput line item and the
        notes for that line item.
     */

   move( @bkdetail.bkno, @bkscrnote.bkno);
   move( @bkdetail.recno, @bkscrnote.recno);
   flexkey = SAVEKEY;
   return(-1);
}


END
```

5. The Simple Report - tbagentr.flx

```
TABLES
  tbagent
END


SCREEN  select frame
{

 DEMO                        Agent Report                        DATE:[tday    ]


       Report Destination:[d        ]  (S=Screen, P#=Printer, D=Disk, A=Aux)
       Report Copies:     [c ]   (1 - 10)

}
END

ATTRIBUTES

tday = displayonly type date, default = today,  noupdate, noentry;
d(rptdest)   = displayonly type character, required, upshift;
c(rptcopies) = displayonly type smallint,  required, include( 1 to 10);

END


SELECT
      tbagent( code)
      EXTRACTALL
END


REPORT

heading
{
DEMO                            Agent Report                          PAGE:[page]
                                                                   [tim     ][tday    ]
================================================================================

  Code   Last Name          First Name Soc Sec No.     Date Hired  Last Raise
  -----  ---------------- ----------- --------------  ----------  ----------
}

detail
{
  [eno ] [elname         ][efname    ][esocno       ] [ehdate  ] [erdate  ]

      Pay Method:  [e]    Salary  [esalary ]   Commission Rate [ecom]%

}

END

ATTRIBUTES

tday = displayonly type date, default = today;
tim  = displayonly type mtime, default = time;
page = displayonly type smallint;

eno     = tbagent.code, right;
esocno  = tbagent.socno;
elname  = tbagent.lname;
efname  = tbagent.fname;
ehdate  = tbagent.hire_date;
erdate  = tbagent.raise_date;
e       = tbagent.paymethod;
esalary = tbagent.salary, format="#,###.##";
ecom    = tbagent.commission, format="###.#";

END
```

6. Sample Output - tbagentr.flx

```
DEMO                              Agent Report                              PAGE:    1
                                                                         17:25:16  01/17/91
================================================================================

  Code   Last Name         First Name Soc Sec No.     Date Hired  Last Raise
 -----  ---------------- ---------- -------------  ----------  ----------
    BF   Favero           Bruce       546907780       02/01/88   01/06/86

     Pay Method:   1     Salary   1,000.99    Commission Rate       %

    DT   Torence          Donna       546907780       04/01/85   01/06/86

     Pay Method:   3     Salary     300.99    Commission Rate       %

    GP   Pollard          Gary        546907780       01/01/85   01/06/86

     Pay Method:   2     Salary     200.99    Commission Rate       %

    HA   Henry            Adams       546907780       02/01/85   01/06/86

     Pay Method:   1     Salary   1,000.99    Commission Rate       %

    JM   Mobley           Janice      546907780       03/01/85   01/06/86

     Pay Method:   1     Salary   1,000.99    Commission Rate       %

    JS   Smith            John        546907780       02/01/85   01/06/86

     Pay Method:   1     Salary   1,000.99    Commission Rate       %

    LR   Renolds          Leon        546907780       04/01/85   01/06/86

     Pay Method:   3     Salary     300.99    Commission Rate       %

    LS   Sanford          Larry       546907780       04/01/85   01/06/86

     Pay Method:   3     Salary     300.99    Commission Rate       %

    MV   Valesques        Maria       546907780       04/01/85   01/06/86

     Pay Method:   3     Salary     300.99    Commission Rate       %

    SJ   Jefferson        Susan       546907780       03/01/85   01/06/86

     Pay Method:   1     Salary   1,000.99    Commission Rate       %
```

7. Advanced Report I - saagent.flx

```
TABLES
  tbagent
  tbven
  bkmaster
  bkdetail
END


SCREEN  select frame
{

 DEMO                    Sales By Agent Report               DATE:[tday    ]


       Report Destination:[d         ]  (S=Screen, P#=Printer, D=Disk, A=Aux)
       Report Copies:     [c ]   (1 - 10)
       Report Title Page: [t]    (Y=Yes, N=No)


       Agent:  1) [agen]   [bempname    ]
               2) [age2]   [bempnam2    ]
               3) [age3]   [bempnam3    ]

       Booking Date Range: [bdate   ]  to  [edate   ]

       Supplier: [sup   ]  [supname         ]

       Sales Only: [s]  (Y=Yes or leave blank)

}
END

ATTRIBUTES

tday = displayonly type date, default = today,  noupdate, noentry;
d(rptdest)   = displayonly type character, required, upshift;
c(rptcopies) = displayonly type smallint,  required, include( 1 to 10);
t(rpttitle)  = displayonly type character, required, upshift, include( Y, N);

agen = bkmaster.agent, upshift,
    lookup(tbagent.tbagentkey, select.bempname = tbagent.lname),
    tablehelp("flex tbagent asdc", tbagent.tbagentkey, tbagent.code, tbagent.lname);

age2 = bkmaster.agent, upshift,
    lookup(tbagent.tbagentkey, select.bempnam2 = tbagent.lname),
    tablehelp("flex tbagent asdc", tbagent.tbagentkey, tbagent.code, tbagent.lname);

age3 = bkmaster.agent, upshift,
    lookup(tbagent.tbagentkey, select.bempnam3 = tbagent.lname),
    tablehelp("flex tbagent asdc", tbagent.tbagentkey, tbagent.code, tbagent.lname);

bempname = displayonly type char, noupdate, noentry;
bempnam2 = displayonly type char, noupdate, noentry;
bempnam3 = displayonly type char, noupdate, noentry;

bdate = bkmaster.bookdate;
edate = bkmaster.bookdate, default = today;

sup = bkdetail.supplier, upshift,
    lookup(tbven.tbvenkey, select.supname = tbven.name),
    tablehelp("flex tbven asdc", tbven.tbvenkey, tbven.code, tbven.name);

supname = displayonly type char, noupdate, noentry;

s  = tbven.saleflag;

END


SELECT

       bkmaster( agent, bookdate, bkno)
           EXTRACTALL

       bkdetail( bkno) bkmaster( bkno)
           EXTRACTALL

       tbven( code)  bkdetail( supplier)
           EXTRACT

       tbagent( code) bkmaster( agent)
           EXTRACT

       WHERE whereselect
```

```
END

REPORT

heading  breakon( bkmaster.agent) newpage everypage pitch12
{
DEMO                    Sales by Agent Report                          PAGE:[page]
                                                                  [tim     ][tday    ]
================================================================================
Agent:[bemp]  [bempname       ]
                                                        Sale
      Book Date Book No Name                           Amount
      --------- ------- ---------------                ----------
}

heading  breakon( bkmaster.bkno)
{
      [bdate   ][bkno  ][bclient             ]     [bsalam     ]
}

detail
{
            [bsup  ] [bship ][bdescription        ][dsalam     ]
}

total  breakon( bkmaster.bookdate,4)
{
                                  ----------------------------
                                   WEEK Subtotal:$[dsalam      ]

}


total  breakon( bkmaster.bookdate,2)
{
                                  ----------------------------
                                  MONTH Subtotal:$[dsalam      ]

}


total  breakon( bkmaster.agent)
{
                                  ----------------------------
                                  AGENT Subtotal:$[dsalam      ]

}


total
{
                                  ============================
                                  GRAND TOTALS:  $[dsalam      ]

}

END

ATTRIBUTES

tday = displayonly type date, default = today;
tim  = displayonly type mtime, default = time;
page  = displayonly type smallint;

bkno = bkmaster.bkno, right;
bclient = bkmaster.name;
bdate= bkmaster.bookdate;
bemp = bkmaster.agent;
bempname = tbagent.lname;
bsalam = bkmaster.salamount, format="#,###.##", nodisplay;

bsup = bkdetail.supplier;
bship = bkdetail.ship;
bdescription = bkdetail.descript;
dsalam= bkdetail.salamount, format="#,###.##";

END
```

8. Advanced Output I - saagent.flx

```
DEMO                    Sales by Agent Report                        PAGE:    1
                                                                  18:00:16  01/17/91
================================================================================
Agent: GP      Pollard
                                                          Sale
      Book Date Book No Name                            Amount
      --------- ------- ---------------                 ----------
       02/02/87       1  gerard                          1,330.00
             CCL      TUG     Cruise Cabin A               300.00
             HOTEL    CLAR    Double Suite                 200.00
             AIR              Regular Faire                100.00
             PCL              Cruise Delux Cabin           150.00
             INSUR            $100,000                      10.00
             SIT      DUKE    Cruise                       250.00
             INSUR            $250,000                      20.00
             HOTEL    DUKE    Regular Faire                300.00
                                  ----------------------------
                                   WEEK Subtotal:$     1,330.00

                                  ----------------------------
                                  MONTH Subtotal:$     1,330.00

       03/03/87       3  gerard                            500.00
             AIR      DUKE    Regular Faire                100.00
             PCL      DUKE    Regular Faire                150.00
             INSUR    DUKE    Regular Faire                250.00
       03/04/87       4  gerard                            500.00
             CCL      DUKE    Regular Faire                100.00
             SIT      DUKE    Regular Faire                150.00
             PCL      DUKE    Regular Faire                250.00
       03/05/87       5  gerard                            500.00
             AIR      DUKE    Regular Faire                100.00
             SIT      DUKE    Regular Faire                150.00
             PCL      DUKE    Regular Faire                250.00
       03/06/87       6  gerard                            500.00
             CCL      DUKE    Regular Faire                100.00
             SIT      DUKE    Regular Faire                150.00
             HOTEL    DUKE    Regular Faire                250.00
       03/07/87       7  gerard                            500.00
             AIR      DUKE    Regular Faire                100.00
             SIT      DUKE    Regular Faire                150.00
             HOTEL    DUKE    Regular Faire                250.00
                                  ----------------------------
                                   WEEK Subtotal:$     2,500.00

                                  ----------------------------
                                  MONTH Subtotal:$     2,500.00

                                  ----------------------------
                                  AGENT Subtotal:$     3,830.00

DEMO                    Sales by Agent Report                        PAGE:    2
                                                                  18:00:16  01/17/91
================================================================================
Agent: GWM
                                                          Sale
      Book Date Book No Name                            Amount
      --------- ------- ---------------                 ----------
       03/08/87       8  menicucci/gerard                  500.00
             PRINCE   DUKE    Regular Faire                100.00
             SITMAR   DUKE    Regular Faire                150.00
             PEARL    DUKE    Regular Faire                250.00
       03/09/87       9  menicucci/gerard                  500.00
             PRINCE   DUKE    Regular Faire                100.00
             SITMAR   DUKE    Regular Faire                150.00
             PEARL    DUKE    Regular Faire                250.00
                                  ----------------------------
                                   WEEK Subtotal:$     1,000.00

                                  ----------------------------
                                  MONTH Subtotal:$     1,000.00

                                  ----------------------------
                                  AGENT Subtotal:$     1,000.00

                                  ============================
                                  GRAND TOTALS:   $     4,830.00
```

9. Advanced Report II - saagent2.flx

```
TABLES
  tbagent
  tbven
  bkmaster
  bkdetail
END


SCREEN  select frame
{

 DEMO                    Sales By Agent Report               DATE:[tday    ]


       Report Destination: [d  ]  (S=Screen, P#=Printer, D=Disk, A=Aux)
       Report Copies:      [c ]   (1 - 10)
       Report Title Page:  [t]    (Y=Yes, N=No)


       Agent:  1) [agen]   [bempname    ]
               2) [age2]   [bempnam2    ]
               3) [age3]   [bempnam3    ]

       Booking Date Range: [bdate   ]  to  [edate   ]

       Supplier: [sup   ]  [supname         ]

       Sales Only: [s]  (Y=Yes or leave blank)

}
END

ATTRIBUTES

tday = displayonly type date, default = today,  noupdate, noentry;
d(rptdest)   = displayonly type character, required, upshift;
c(rptcopies) = displayonly type smallint,  required, include( 1 to 10);
t(rpttitle)  = displayonly type character, required, upshift, include( Y, N);

agen = bkmaster.agent, upshift,
    lookup(tbagent.tbagentkey, select.bempname = tbagent.lname),
    tablehelp("flex tbagent asdc", tbagent.tbagentkey, tbagent.code, tbagent.lname);

age2 = bkmaster.agent, upshift,
    lookup(tbagent.tbagentkey, select.bempnam2 = tbagent.lname),
    tablehelp("flex tbagent asdc", tbagent.tbagentkey, tbagent.code, tbagent.lname);

age3 = bkmaster.agent, upshift,
    lookup(tbagent.tbagentkey, select.bempnam3 = tbagent.lname),
    tablehelp("flex tbagent asdc", tbagent.tbagentkey, tbagent.code, tbagent.lname);

bempname = displayonly type char, noupdate, noentry;
bempnam2 = displayonly type char, noupdate, noentry;
bempnam3 = displayonly type char, noupdate, noentry;

bdate = bkmaster.bookdate;
edate = bkmaster.bookdate, default = today;

sup = bkdetail.supplier, upshift,
    lookup(tbven.tbvenkey, select.supname = tbven.name),
    tablehelp("flex tbven asdc", tbven.tbvenkey, tbven.code, tbven.name);

supname = displayonly type char, noupdate, noentry;

s  = tbven.saleflag;

END


SELECT

       bkmaster( bookdate, bkno)
             bookdate
             agent
           EXTRACTALL

       bkdetail( bkno)    bkmaster( bkno)
              salamount
           EXTRACTALL

       tbagent( code)   bkmaster( agent)
              fname
              lname
           EXTRACT
```

```
        WHERE whereselect

        ONINDEX bktemp( agent, bookdate)

END


SELECT

        bktemp( agent, bookdate)
            EXTRACTALL

END


REPORT

heading  breakon( bktemp.agent) pitch12
{
DEMO                    SALES ANALYSIS REPORT                        PAGE:[page]
                        Sales Summary by Agent              [tim     ][tday    ]
================================================================================


                              Sales
Agent                         Amount
---------------------------   ----------------
}

heading  breakon( bktemp.agent)
{
[agent ][fname    |lname          ]
}


total  breakon( bktemp.bookdate,2)
{
           Month of[bd]:    $[dsalam        ]

}


total  breakon( bktemp.agent)
{
                              ---------------
      SUBTOTAL by Reference:$[dsalam        ]


}

total
{

               GRAND TOTALS:$[dsalam        ]

}

END


ATTRIBUTES

tday  = displayonly type date, default = today;
tim   = displayonly type mtime, default = time;
page  = displayonly type smallint;

bd= bktemp.bookdate;
agent = bktemp.agent;
fname = bktemp.fname;
lname = bktemp.lname;

dsalam= bktemp.salamount, format="#,###.##";

END
```

10. Advanced Output II - saagent2.rpt

```
DEMO                  SALES ANALYSIS REPORT                        PAGE:    1
                      Sales Summary by Agent                 18:04:08  01/17/91
================================================================================


                                Sales
Agent                           Amount
---------------------------   ----------------
 GP      Gary      Pollard
           Month of 02 :    $       1,330.00

           Month of 03 :    $       2,500.00

                             ---------------
      SUBTOTAL by Reference:$       3,830.00


DEMO                  SALES ANALYSIS REPORT                        PAGE:    1
                      Sales Summary by Agent                 18:04:08  01/17/91
================================================================================


                                Sales
Agent                           Amount
---------------------------   ----------------
 GWM
           Month of 03 :    $       1,000.00

                             ---------------
      SUBTOTAL by Reference:$       1,000.00


               GRAND TOTALS:$       4,830.00
```

11. The Menus - menu.flx

```
TABLES
  sysfile
  menuhead
  menufield
  menuuser
  menuusty
  menuperm
END


MENU demomenu
{



                @_____________________________________________@
                |     Welcome to the INFOFLEX DEMO !!!        |
  @_____________|_____________________________________________|_____________
  | This sample application has been designed to demonstrate the features    |
  | available with Infoflex-4GL.  You should refer to Appendix 1 in the User |
  | Guide for instructions on how to proceed thru the DEMO. Source listings  |
  | can also be found in the Appendix. Note that this entire DEMO runs from  |
  | a single 301K executable and uses only 40 lines of procedural code !!!   |
  |__________________________________________________________________________|
                [m1                                  ] *(8,11)
                [m2                                  ] *(8,11)
                [m3                                  ] *(3,3)
                [m4                                  ] *(4,4)
                [m5                                  ] *(6,5)
                [m6                                  ] *(7,7)
                [m7                                  ]
                 * User Guide reference (Chapter, Appendix topic)

                       Enter Selection > [s ]
}
END


ATTRIBUTES

m1 = menufield.itemdesc;
m2 = menufield.itemdesc;
m3 = menufield.itemdesc;
m4 = menufield.itemdesc;
m5 = menufield.itemdesc;
m6 = menufield.itemdesc;
m7 = menufield.itemdesc;
s = displayonly type char;

END


MENU bookmenu
{



   @_____________________________________ @_____________________________________
   |        DATA ENTRY PROGRAMS          | |            REPORTS                  |
   |_____________________________________| |_____________________________________|
       [m1                                ]    [m11                                ]
       [m2                                ]    [m12                                ]
       [m3                                ]    [m13                                ]
       [m4                                ]    [m14                                ]
       [m5                                ]    [m15                                ]
       [m6                                ]@_____________________________________
       [m7                                ]|    END-OF-THE MONTH PROCESSING     |
       [m8                                ]@_____________________________________|
       [m9                                ]    [m16                                ]
   @_____________________________________       [m17                                ]
   |          QUERY ON-LINE              |      [m18                                ]
   |_____________________________________|
       [m10                               ]


                       Enter Selection > [s ]
}
END


ATTRIBUTES

m1 = menufield.itemdesc;
```

```
m2 = menufield.itemdesc;
m3 = menufield.itemdesc;
m4 = menufield.itemdesc;
m5 = menufield.itemdesc;
m6 = menufield.itemdesc;
m7 = menufield.itemdesc;
m8 = menufield.itemdesc;
m9 = menufield.itemdesc;
m10 = menufield.itemdesc;
m11 = menufield.itemdesc;
m12 = menufield.itemdesc;
m13 = menufield.itemdesc;
m14 = menufield.itemdesc;
m15 = menufield.itemdesc;
m16 = menufield.itemdesc;
m17 = menufield.itemdesc;
m18 = menufield.itemdesc;
s = displayonly type char;

END


MENU  devmenu
{




      MENU                              PROGRAMMING
          [m1                    ]          [m11                   ]
          [m2                    ]          [m12                   ]
          [m3                    ]          [m13                   ]
          [m4                    ]          [m14                   ]
      SECURITY                              [m15                   ]
          [m5                    ]          [m16                   ]
          [m6                    ]          [m17                   ]
          [m7                    ]      MISCELLANEOUS
          [m8                    ]          [m18                   ]
          [m9                    ]          [m19                   ]
          [m10                   ]          [m20                   ]
                                            [m21                   ]
                                            [m22                   ]
                                            [m23                   ]


                          Enter Selection > [s ]
}
END


ATTRIBUTES

m1 = menufield.itemdesc;
m2 = menufield.itemdesc;
m3 = menufield.itemdesc;
m4 = menufield.itemdesc;
m5 = menufield.itemdesc;
m6 = menufield.itemdesc;
m7 = menufield.itemdesc;
m8 = menufield.itemdesc;
m9 = menufield.itemdesc;
m10 = menufield.itemdesc;
m11 = menufield.itemdesc;
m12 = menufield.itemdesc;
m13 = menufield.itemdesc;
m14 = menufield.itemdesc;
m15 = menufield.itemdesc;
m16 = menufield.itemdesc;
m17 = menufield.itemdesc;
m18 = menufield.itemdesc;
m19 = menufield.itemdesc;
m20 = menufield.itemdesc;
m21 = menufield.itemdesc;
m22 = menufield.itemdesc;
m23 = menufield.itemdesc;
s = displayonly type char;

END
```

12. Miscellaneous Screen I - sysfile.flx

```
tables
  sysfile
end

select
   sysfile( sysname)
   EXTRACTALL
end

screen  sysfile frame
{

 INFOFLEX  [modemsg      ] SYSTEM FILE MAINTENANCE SCREEN     DATE:[today    ]


     System Name: <[sysname    ]>

     Company Name: [company                                  ]


}
end

attributes

modemsg = displayonly type character, noupdate, noentry, reverse, retain;
today = displayonly type date, noupdate, noentry, default=today;

sysname = sysfile.sysname, upshift,
      required, comments = "Enter system abbreviation";

company = sysfile.company,
   comments = "Enter company name";

end
```

13. Miscellaneous Screen II - tbven.flx

```
TABLES
  tbven
END


SELECT
   tbven(code)
   EXTRACTALL
END


SCREEN  tbven frame defaulton
{

 DEMO       [modemsg      ]      Supplier Entry Screen         DATE:[tday    ]



                Code <[vno   ]>

                Name:   [vname               ]

                Contact:[contact             ]

                Phone:  [phone          ]

                Type  [t]

                Sale Flag [s]


}
END


ATTRIBUTES

modemsg = displayonly type character, noupdate, noentry, reverse, retain;
tday = displayonly type date, noupdate, noentry, default=today;

vno = tbven.code, upshift,
      required, comments =
"Enter the supplier's code to identify this supplier throughout the system";

vname = tbven.name,
   comments = "Enter supplier's name";

contact = tbven.contact;
phone   = tbven.phone;

t = tbven.type, upshift, required, include( T, C, O),
    comments = "Enter supplier type: T= Tour, C=Cruise, O=Other";

s = tbven.saleflag, upshift, required, include( Y, N),
    comments = "Sale included in Sales Analysis Reports: Y=Yes, N=No";

END
```

INDEX

INDEX

INFOFLEX®, INC

I N F O F L E X - 4 G L

Reference Guide

Printed in U.S.A. on May 2006

TABLE OF CONTENTS

1. INTRODUCTION

Infoflex In Brief

Infoflex is an computer application development language and environment built around an Informix™ compatible relational database. Infoflex specializes in the rapid development and integration of application menus, screen forms, and reports.

Scope

The Reference Manual will provide a concise explanation of Infoflex features.

Infoflex is targeted for the UNIX or XENIX and DOS operating systems. Porting to other operating systems will done upon request. This manual assumes that you have a working knowledge of your operating system.

Manual Organization

The Reference Manual is organized in chapters, and each chapter is subdivided into topics. A topic is always headed with an emboldened title unindented to the left of the main body of text. For example, this topic is **Manual Organization** within the chapter, **Introduction**.

The Reference Manual may further subdivide its topics into subtopics, which also have emboldened titles. Subtopics are indented within topics so that they are easily recognized. The intent is for you to find things quickly in the Reference Manual.

2. DEVELOPMENT PROCEDURES

Software Installation

Certain software must be in place before you begin your Infoflex application development. The following subtopics will describe these.

Infoflex

The Infoflex installation procedure is concisely laid out for each hardware and operating system configuration and media of distribution. Each release of the Infoflex will include an instruction sheet enclosure.

The Infoflex Development Package is made up of several sub-packages. Each sub-package may be installed independent of the others or in any combination.

SCREENFLEX. If your applications use screen forms you will need SCREENFLEX. The features and language of SCREENFLEX are detailed in Chapter 3 and 4.

REPORTFLEX. If your applications use reports you will need REPORTFLEX. The features and language of REPORTFLEX are detailed in Chapter 5.

MENUFLEX. If your applications use menus you will need MENUFLEX. The features and language of MENUFLEX are detailed in Chapter 6.

SQLFLEX. SQLFLEX is the Infoflex language and procedure for creating and modifying the structure of your Infoflex databases and querying their contents. The features and language of SQLFLEX are detailed in Chapter 12.

The following Infoflex packages are not stand-alones but must be incorporated with one or more of the stand-alone packages above:

TOOLFLEX. Once you begin linking C language code in with your Infoflex screens and reports, you may wish to make use of the special C function library available under TOOLFLEX. These functions assist in the data manipulation between an Infoflex database and your application screens and reports. The functions of TOOLFLEX are detailed in Chapter 10.

PRINTFLEX. PRINTFLEX is included with all Infoflex installations. It provides the means of setting up the printer characteristics, such as font changes, for any printer used with Infoflex.

C Compiler

If you intend to link in your own C functions with your Infoflex applications you will need a C language development system. On UNIX the compiler should be in the standard directories. On DOS the C compiler is expected to be under the **\c** directory.

D-ISAM™ or C-ISAM™

If you use an **INSTRUCTIONS** section anywhere in your in application, you will need to install the C language library for the D-ISAM file manager from Byte Designs or the C-ISAM file manager from Informix. Follow their installation instructions exactly to achieve compatibility with the Infoflex development system. On UNIX the libraries are expected to be in the /.../**fx/lib** directory and on DOS the \...**fx\lib** directory.

Application Set-Up

Once your Infoflex Development Package and supporting software are installed, you are ready to initiate the development of your application. The first step in preparing your development environment is to set up the Infoflex environment variables. Chapter 9 describes the various environment variables and their purpose. To assist with setting these variables, we provide a utility called **fxsetenv**. The **fxsetenv** utility is run slightly different on UNIX versus DOS.

ON UNIX:

Set **FXDIR** to the Infoflex base directory, **FXAPDIR** to the application's base directory, and then execute the **fxsetenv** utility. For example,

```
$ FXDIR=/.../fx; export FXDIR
$ FXAPDIR=/.../appname; export FXAPDIR
$ .   $FXDIR/unx/fxsetenv
```

ON DOS:

Execute the **fxsetenv** utility with the Infoflex base directory and the application base directory as arguments. For example,

```
> \fx\dos\fxsetenv    \...\fx    \appname
```

Once the environment variables are properly set, the next step is to create your application directory and database structure. To create your application structure run the **fxmkapp** utility as follows:

ON UNIX:

The **fxmkapp** utility takes the Infoflex base directory as the first argument and the application base directory as the second argument. The **fxmkapp** will do the following:

```
$ fxmkapp    /.../fx     /.../appname
```

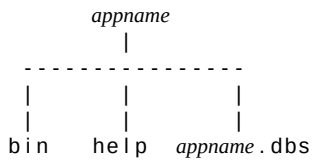
ON DOS:

The **fxmkapp** utility takes the Infoflex base directory as the first argument and the application name as the second argument.

```
> fxmkapp   \...\fx      appname
```

The **fxmkapp** will do the following:

- Create the following directory structure:



  The **bin** directory will be where you will develop and compile the Infoflex source files of your application. The **help** directory will be where your on-line help files are stored. *Appname***.dbs** is your application's database directory.

- Install the Infoflex development menu system (MENUFLEX) in the application **bin** directory. From the development menu you may create and compile Infoflex source files, **.flx** files, and link in any C functions you have written for your application. The development menu also has an option to edit the menu tables, that define the menu structure of the application, and an option to edit the **sysfile** table, that defines the company and application names to appear in the menu headers.
- The **sysmsg.flx** file is placed in the **bin** directory. This file contains all the standard run-time messages of an Infoflex application and will interface with your application automatically with no special programming. This file also defines the default function key assignments, function key label names, and color parameters.

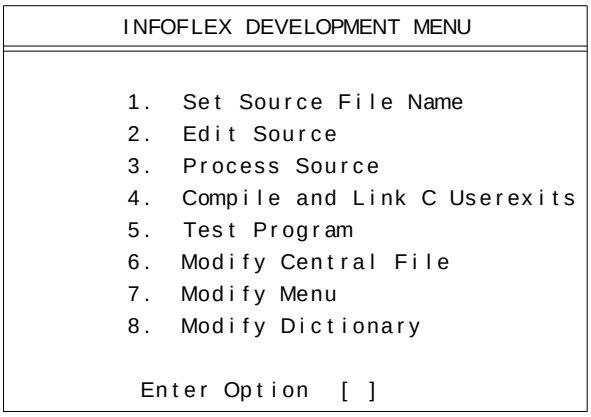
- Create two executable files in the application **bin** directory with the names *appname* and *xappname*. *Appname* will set the appropriate application environment variables and start-up your application menu system. *Xappname* will set the appropriate developmental environment variables and start up the development menu.

The Development Menu

This topic describes the options of the Infoflex development menu. If you did not purchase the MENUFLEX subpackage, you will not be able to develop from the menu system. Instead, you should follow the instructions in the next topic on how to develop from the command line.

Before starting, move to the **bin** directory of the application that you wish to develop. Execute the *xappname* program to call up the development menu. The *xappname* program was created by the **fxmkapp** utility when setting up the application.

The menu will appear as follows:



This is briefly what each menu option does.

1. Set Source File Name

You will be prompted for the source file name without the **.flx** extension. The name should be eight characters or less for compatibility with DOS. The name you enter will be displayed in the upper right hand corner of the video display.

2. Edit Source

You will be placed into the editor to create or modify an Infoflex source file (the environment variable **FXEDIT** determines which editor). The content of this source file will define a menu, screen, or report layout and its field attributes. See Chapter 3 through 7 for a detailed description of the contents of the Infoflex source file. One requirement of the development menu system is, if you wish to test your application from the menu, that there be only one **SCREEN** section per screen source file and one **REPORT** section per report source file.

3. Process Source

Your source file will be compiled into a loadable Infoflex **.pic** file.

4. Compile and Link C Userexits

Here you may compile and link any C language code generated by Step 3. C code will be generated if there is an **INSTRUCTIONS** section in the source file. The resultant executable program will have the same name as the source file but without a file name extension.

5. Test Program

You may run your compiled menu, screen form, or report from here.

6. Modify Central File

This option allows you to update the system wide information, specifically the application title and the company name, using a canned Infoflex screen form.

7. Modify Menu

This option allows you to update the menu structure of your application using the **cwmenu** utility (see Topic 6.1). Note that the development menu definition itself is stored in the menu tables of your application database.

8. Modify Dictionary

You will be placed in the editor to create an SQL script. The SQL script may modify the application database structure or query the database. Once you have saved your script, the SQLFLEX utility is invoked to process your SQL script.

Development Commands

This topic will show you how to develop an Infoflex program without using the development menu system.

Developers wanting to use a makefile, a UNIX automated compilation utility, or who feel more comfortable developing from the operating system command line will want to read this chapter.

The first step is to set the environment variables using the **fxsetenv** utility described in previous topic on Application Set-Up. Once the environment variables are set you are ready to do any of the following developmental procedures:

1. Set Source File Name

This step is not used from the operating system command line.

2. Edit Source

Use the text editor of your choice to create an Infoflex source file. The file name has the form: *filename***.flx**.

3. Process Source

Use the **fxpp** command to compile your Infoflex source file.

fxpp *filename*

filename without the **.flx** extension, produces the file *filename***.pic** that is loaded at application run-time to produce your menu, screen form, or report.

4. Compile and Link C Userexits

If your source file has an **INSTRUCTIONS** section, **fxpp** will also produce the file *filename***.c**, which is the **INSTRUCTIONS** section code fully translated to C language code. In this case you will need to further compile *filename***.c** with:

fxcl *filename.c* -o *progname*

The final result is the executable file *progname*.

5. Test Program

In the case where there is no **INSTRUCTIONS** section associated with your *filename***.pic** file, you execute *filename***.pic** with:

flex *filename*

In the above example where we compiled and linked in **INSTRUCTIONS** section C code and produced the executable file *progname*,

progname filename

becomes the command for executing *filename***.pic**.

Also see Topic 3.7, RUNNING THE SCREEN FORM.

6. Modify Central File

The command for modifying the central file is:

flex sysfile

7. Modify Menu

The command for modifying your application's menu data is:

flex cwmenu

See Topic 6.1, THE CWMENU UTILITY.

8. Modify Dictionary

To modify the structure of your application database structure, you must create with a text editor an SQLFLEX script of the form: *script***.sql**. This script will contain database modifying SQLFLEX statements that are executed with the command:

fxsql *appname script*

You must be in the your application directory *appname* when running this command.

In the case that you do not have the Infoflex SQLFLEX sub-package, you will need to use Informix-SQL or a similar database management system to modify your database. Refer to the documentation for the system you are using.

Advanced Development Techniques

This topic covers more advanced development techniques for reducing processing time, combining programs, and using makefiles. These techniques are primarily useful for Infoflex programs containing **INSTRUCTIONS** section C language code.

Reducing Processing Time

The **fxpp** command offers a command line option for selectively processing the **INSTRUCTIONS** section. By using this command option, you can avoid processing the entire **.flx** file when there has only been a change to the **INSTRUCTIONS** section. Here is an example command line that will process the **INSTRUCTIONS** section only.

```
fxpp -c filename
```

This will generate the *filename***.c** file and leave the *filename***.pic** alone.

When using the technique you need to be fairly confident that the **INSTRUCTIONS** section is the only part of your source file that has changed. One way to solve this problem is to move the **INSTRUCTIONS** section outside of the file containing the screen/report definition. If the **INSTRUCTIONS** are in a separate file or files, then it is easier to determine from the file dates what parts need recompiling. You can even go a step further and use an automated recompliation technique, such as a makefile system, that determines automatically which files need recompilation and which files are already up to date. More on makefiles later.

Combining Programs

In this subtopic we discuss how to combine your Infoflex executable programs. With little effort it is possible to combine your entire application into a single executable program.

What often happens after developing a sophisticated application is you end up with a number of source files that use **INSTRUCTIONS** section code. Each of these source files, when fully compiled, result in a sizeable executable program. This is often undesirable for three reasons: disk space is used up, response time for switching programs is slow, and run-time memory is used up (especially for multi-user installations). Infoflex offers an easy way of combining the executable programs to avoid these disadvantages.

The steps for combining programs are:

1. Process each **.flx** file that is to be combined using the '-n' option as follows:

```
fxpp -n filename1
fxpp -n filename2
fxpp -n filename3
   .
   .
   .
```

2. Create a file *filelist.lst* containing a list of all **.flx** files that will be sharing the same executable. You may have duplicate C language function names across these **.flx** files as long as they are declared as **static** (this is applies to main() functions as well). *Filelist.lst* should look as follows:

```
filename1.flx
filename2.flx
filename3.flx
   .
   .
   .
```

3. Process the *filelist***.lst** as follows:

```
fxpp -l filelist.lst
```

This process will generate a C source file *filelist*.**c** that will later be compiled and linked with the executable.

4. Compile all **.c** files generated from step 1 and 3 above. Note that **.c** files will only be generated for **.flx** files having an **INSTRUCTIONS** section.

   fxcl -c *filelist*.c
   fxcl -c *filename1*.c
   fxcl -c *filename2*.c
   .
   .
   .

5. Link object code generated in step 4 together into a single executable (**.o** is **.obj** on DOS):

   fxcl -o *progname filelist*.o *filename1*.o *filename2*.o ...

6. Change the menu option to load a screen internally. This is done by changing the TYPE to **F** and having the execution line call the **flex** function.
7. Initiate the application using the *progname* on the command line.

The Makefile

A makefile system will automate your compilation activity, and for large multi source file applications, it is almost imperative that you use makefiles. A makefile will contain the rules and steps for creating your application from source files. Such a system will also enable you to reconstruct the application at any time with the minimum recompilation steps.

On UNIX and XENIX the makefile interpreter program called **make** comes with the standard C language development system. On DOS **make**-like utilities are available for program development. For a thorough understanding of the theory, syntax, and usage of makefiles, we refer you to the documentation that comes with your particular makefile system.

Here is a practical example using a makefile. Our application module is made up of three source files:

```
scrn.flx
uexits1.flx
uexits2.flx
```

Scrn.flx contains only the screen definition, **uexits1.flx** and **uexits2.flx** contain the **INSTRUCTIONS** section code.

The following will be the UNIX or XENIX makefile text for our example above:

```
.SUFFIXES: .o .flx

.flx.o:
        fxpp -p scrn.pic -n -c $*
        fxcl -c $*.c
        rm *.c

OBJS = scrn.o uexits1.o uexits2.o

applic: $(OBJS)
        fxcl -o applic $(OBJS)

scrn.o: scrn.flx
        fxpp -x scrn
        fxpp -l scrn.lst
        fxcl -c scrn.c
        rm -f scrn.c $(OBJS)
```

We consider this the optimum makefile structure for an Infoflex application, and all Infoflex development procedures can be patterned after it. The new **fxpp** options used here are documented in the next subtopic.

If you are not familiar with UNIX makefiles (and a good DOS makefile system should pattern itself after the UNIX model), then I would not go any further and would first familiarize yourself with your makefile system. Further explanation here assumes that you at least understand the makefile syntax and sematics in the above example.

As a further optimization, the makefile will create the userexits array as a separate **.c** file. The command that accomplishes this is:

```
fxpp -l scrn.lst
```

where the contents of **scrn.lst** is simply:

```
scrn.flx
```

The next step is to compile the resultant **scrn.c** with:

```
fxcl -c scrn.c
```

So the rationale is, if we update **scrn.flx**, a new **scrn.pic** and **scrn.o** will be generated. We then remove **scrn.c** as it is an intermediate file that we do not need to keep. We also remove **uexits1.o** and **uexits2.o** (**rm -f** is used so that if these files are missing, **rm** does not generate an error), which forces the recompilation of these two **.o** files. The idea is that since **scrn.pic** has been regenerated, the **INSTRUCTIONS** code in **uexits1.flx** and **uexits2.flx**, which depends on the content of **scrn.pic**, must be recompiled. The general **.flx.o** rule of the makefile recompiles **uexits1.flx** and **uexits2.flx**.

Now we can see that the minimum compilation steps are taken with each regeneration of the application **applic**. If **scrn.flx** is modified, the **.pic** and all **.o**'s are regenerated. If only **uexits1.flx** is modified, then only **uexits1.o** is regenerated.

The command:

```
make
```

looks for the file **makefile** or **Makefile** in the current directory and processes it.

FXPP Options

-x This option wll generate the **.pic** file but not process the source file's **INSTRUCTIONS** section that would produce the **.c** file.

-c Use this option to process just the **INSTRUCTIONS** section to generate the **.c** file. The **.pic** file will not be generated.

-n Normally the **.c** file built from an **INSTRUCTIONS** section includes an array variable listing the userexit names. The **.c** file generated when using the **-n** option excludes this userexits array. The **.pic** file will be generated however.

-p Use this option with the additional argument *filename***.pic**. Here **fxpp** will not look to the source file being compiled for the **.pic** information, but instead reads an earlier compiled *filename***.pic**. This would be one way to compile **INSTRUCTIONS**-only source files.

-l Use this option with the additional argument *filename***.lst** which is a text file containing a vertical list of **.flx** file names. **Fxpp** will then generate a single **.c** file with a userexits array that is the union of all the userexits of all the source files in the list. No other code is placed in this file except the maximum memory required at run-time to load any one of the **.pic** files compiled from the **.flx** files listed.

Special Processor Commands

The Infoflex processor allows special processor commands, which are lines beginning with the character **#**.

The current processor commands are:

#include insert text from another file.

The **#include** macro causes the entire contents of a specified Infoflex source file to be processed as if those contents had appeared in place of the **#include** macro. The form of the command is:

```
#include "filename"
```

#include macros appearing in the body of the **INSTRUCTIONS** section will not be processed by Infoflex unless the file extension is **.flx**. This is to avoid conflict with the C preprocessor which is run at a later step.

3. SCREENFLEX

SCREENFLEX is the special language for Infoflex screen form development.

Throughout this chapter we will be referring to header and array portions of the screen form. The header portion is where a single record is displayed and updated. The array portion is where multiple records from the same table are displayed and updated. A screen may have a header only, an array only, or have both portions.

3.1 SECTION ORGANIZATION

A SCREENFLEX source file is an ordinary text file of the form *filename*.**flx**. Its content is subdivided into **sections**. The organization of these **sections** has certain ordering rules.

TABLES Section

The **TABLES** section is a required section and must be the first section of the screen source file. Topic 3.2 contains a full description of the SCREENFLEX **TABLES** section.

SELECT Section

The **SELECT** section is required and immediately follows the **TABLES** section. Topic 3.3 contains a full description of the SCREENFLEX **SELECT** section.

SCREEN Section

The **SCREEN** section is required and immediately follows the **SELECT** section. Topic 3.4 contains a full description of the SCREENFLEX **SCREEN** section.

ATTRIBUTES Section

The **ATTRIBUTES** section is required and immediately follows the **SCREEN** section. Topic 3.6 contains a full description of the SCREENFLEX **ATTRIBUTES** section.

INSTRUCTIONS Section

The **INSTRUCTIONS** section is an optional last section of the screen form source file. It contains your C language functions called by any userexits of the screen forms. Chapter 7 contains a full description of the **INSTRUCTIONS** section.

Multiple Screen Definitions

More than one screen form can be defined in a single screen source file. There are a few variations on how the sections can be organized, and those variations will be taken up under the appropriate section topic.

3.2 TABLES SECTION

Overview

The **TABLES** section lists the names of any tables referenced in the screen form source file.

Syntax

```
TABLES
    tablename  [  tableparms  ]
        .
        .
        .
END
```

Description

TABLES	is a required keyword.
tablename	is a database table name.
tableparms	are optional and may be any one of the following: **open**, **read**, **alias** *aliasname*. **Open** parameter will open the file upon starting the program. **Read** parameter will open the file, read the first record, and close the file upon starting the program (usefile for control files). The **alias** *aliasname* parameters will allow you to open the file under a different name.
END	is a required keyword.

Notes

- There can be any number of *tablename*s in the **TABLES** section. There must be at least one.
- In multiple screen source files there can be a separate **TABLES** section before each **SELECT** section, or there can be a single **TABLES** section that includes all the tables referenced in the source file. For example, this would be the outlines of two possible screen source files with three screen definitions:

Outline A

TABLES section

SELECT section
SCREEN section
ATTRIBUTES section

SELECT section
SCREEN section
ATTRIBUTES section

SELECT section
SCREEN section
ATTRIBUTES section

Outline B

TABLES section
SELECT section
SCREEN section
ATTRIBUTES section

TABLES section
SELECT section
SCREEN section
ATTRIBUTES section

TABLES section
SELECT section
SCREEN section
ATTRIBUTES section

- In Outline B here is no problem in specifying the same table in two or three of the **TABLES** sections. It is not an error and will not require redundant memory resources.

Example

```
TABLES
     tbemp
     tbven
     bkmaster
     bkdetail
END
```

3.3 SELECT SECTION

Overview

The optional **SELECT** section defines which database records are to be selected and saved by the Screen Form. This section will also specify the ordering of the records and, when multiple tables are accessed, how they are joined. If the **SELECT** section is not specified, no records will be selected and the user will be placed in **PROMPT** mode (for further explanation on modes see the topic on *Running the Screen Form*).

Syntax

```
SELECT
        mastertable ( sortfields )
                EXTRACTALL
    [ jointable ( joinfields )    mastertable ( mastfields )
                EXTRACTALL
    ]
END
```

Description

SELECT	is a required keyword.
mastertable	is the required primary table from which records will be selected.
sortfields	is the field or fields that comprise an index of *mastertable*. The selected records are ordered by this index. If there are two or more fields, *sortfields* is a comma separated list.
jointable	is an optional table joined to the primary table.
joinfields	is the field or fields that comprise the joined index of *jointable*. The selected *jointable* records are ordered by this index. If there are two or more fields, *joinfields* is a comma separated list.
mastfields	are the fields of the primary table that are joined to *joinfields* of the joined table.
EXTRACTALL	is a keyword required by both table selects. It specifies that the table selection should retrieve all matching records.
END	is a required keyword.

Notes

- The optional *jointable* is only needed when the screen form contains a header portion and an array portion.
- The array portion is the multiple record scrolling region of the screen form. This region is specified by the **REPEAT** and **ENDREPEAT** keywords (see Topic 3.6, ATTRIBUTES SECTION).
- As a record is selected from *mastertable*, the value of its *mastfields* determines the values given to the corresponding *joinfields* in selecting records from *jointable*. The data type of each field of *mastfields* must be the same as the data type of the corresponding *joinfields* field.

Example 1

This is the **SELECT** section for a screen form with a single table defined.

```
SELECT
     bkmaster( bookdate, bkno)
          EXTRACTALL
END
```

For a further examples, see Example 2 under Topic 3.6, ATTRIBUTES SECTION.

Example 2

This is the **SELECT** section for a screen form with both a header and an array portion:

```
SELECT
     bkmaster( bookdate, bkno)
          EXTRACTALL
     bkdetail( bkno)  bkmaster( bkno)
          EXTRACTALL
END
```

3.4 SCREEN SECTION

Overview

The **SCREEN** section defines the layout of the screen form.

Syntax

```
SCREEN screenname
        [ WINDOW( row , col  [ , height , width ) ]  ]
        [ BOX | FRAME ]   [ POPUP ]
        [ zoomscreen( "scrcommand" ) ]
        [ azoomscreen( "ascrcommand" ) ]
        [ joinon(table.indexname, parts) ]
        [ ajoinon(atable.indexname, aparts) ]
        [ userexit( funcname ) ]  .  .  .
{
    literals  [fieldtag        ]  .  .  .
              .
              .
              .
}
END
```

Description

SCREEN	is a required keyword.
screenname	is required and should represent a unique identifier among all **SCREEN** sections of the source file.
WINDOW	is an optional keyword. The **WINDOW** clause specifies alternate video display coordinates where the screen layout will be placed at run-time.
row	Where there is a **WINDOW** clause, this is the top row coordinate. The first row is row 0.
col	Where there is a **WINDOW** clause, this is the leftmost column coordinate. The first column is column 0.
height	Where there is a **WINDOW** clause, this is the optional height or number of rows.
width	Where there is a **WINDOW** clause, this is the optional width or number of columns.
BOX	is an optional keyword that specifies that the screen region defined by **WINDOW** will have a single line border.
FRAME	is an optional keyword that specifies that the screen region defined by **WINDOW** will have a single line border with an additional line separating a title region at the top of the **WINDOW**.
POPUP	is an optional keyword that specifies that the screen will overlay any pre-existing screen.
zoomscreen	is an optional keyword to specify a screen to invoke when the **ZOOM** function key is pressed anywhere within *screenname'*s non-array portion.
scrcommand	is required with a **zoomscreen** clause and is a command line for running a screen. The **zoomscreen** command line begins with the *layout* parameter as specified in topic 7 of this chapter.
azoomscreen	is an optional keyword to specify a screen to invoke when the **ZOOM** function key is pressed anywhere within *screenname 's* array portion.
ascrcommand	is required with an **azoomscreen** clause and is a command line for running a screen. The **azoomscreen** command line begins with the *layout* parameter as specified in topic 7 of this chapter.
joinon	is an optional keyword to specify an index to be relationaly joined to the screen headers index (as specified in the **SELECT** section).

table.indexname is required with the **joinon** clause and is the name of the index that will be relationaly joined to the screen header's index. The values in this index's fields will be related to each field of the screen header's index, therefore, these fields must be of the same type and length.

parts is optional with the **joinon** clause and is the significant number of fields within the *table.indexname* which will be relationaly joined to the screen header's index. The default is all parts of the index.

ajoinon is an optional keyword to specify an index to be relationaly joined to the screen array's index (as specified in the **SELECT** section).

atable.indexname is required with the **ajoinon** clause and is the name of the index that will be relationaly joined to the screen array's index. The values in this index's fields will be related to each field of the screen array's index, therefore, these fields must be of the same type and length.

aparts is optional with the **ajoinon** clause and is the significant number of fields within the *atable.indexname* which will be relationaly joined to the screen array's index. The default is all parts of the index.

userexit is optional and is a userexit name.

funcname is required with a userexit clause and is the name of a C function that you have written in the screen form's **INSTRUCTIONS** section. *Userexit* will pass program control to this function.

{ } are required brackets and enclose the screen layout. Each bracket must be on a line by itself.

literals is that part of the screen layout that will display at run-time exactly as it is represented in the layout.

[ ] are brackets actually used in the screen layout definition and define the position and length of a given field.

fieldtag is a label or name for an input or display field of the screen form. This is the name that is used to reference the field in the subsequent **ATTRIBUTES** section and optional **INSTRUCTIONS** section.

END is a required keyword.

Notes

- All of the fields making up the primary index, as specified in the **SELECT** section, must be defined on the screen.
- Without the *height* parameter for the **WINDOW** clause, the height of the window will be calculated from the screen layout. Without the *width* parameter for the **WINDOW** clause, the width of the window will also be calculated from the screen layout. Without the **WINDOW** clause altogether, **WINDOW (0, 0)** is assumed.
- When exiting a screen entered using **zoomscreen** or **azoomscreen**, *screenname* will be refreshed.
- **Zoomscreen** and **zoomkey** keywords are mutually exclusive as are the keywords **azoomscreen** and **azoomkey**
- To save space used by the field brackets, **[ ]**, you may use the vertical bar **|** to mark the end of one field and the beginning of another.
- Field brackets or field delimiting vertical bars will not appear in the screen form at run-time.
- You may only have *literals* and displayable fields on the first twenty lines of the video display. At run-time SCREENFLEX reserves the bottom three lines for comments, function key labels, prompts, and messages.
- There can be any number of userexit clauses, and if more than two, they are space or newline separated. The specific userexits available to the **SCREEN** section are taken up in the next topic, SCREEN USEREXITS.
- Only the fields of the first row of a screen array are defined in the screen layout. See Topic 3.6, ATTRIBUTES SECTION, for further syntax on the specification of screen arrays.
- For a multi screen form any number of **SCREEN** sections can follow a single **SELECT** section. Each **SCREEN** section would have its own **ATTRIBUTES** section. A requirement is that the *screenname* of the first screen is a prefix of the *screenname*s of the other screen forms. The names **tbemp** and **tbemp2** are an example of the first and second *screenname*s of a multi screen form. A further requirement is that none of the screen layouts of a multi screen form contains a screen array.
- There is a special screen *fieldtag* reserved for displaying the current mode (ADD,CHANGE, etc.). The *fieldtag* is **modemsg** and has a length of 13.

Example

```
SCREEN  bkinput  WINDOW( 4,13,15,65)
        beforesection( startup)
        beforesave( checkval)
{
               BOOKING ENTRY SCREEN
=====================================================

Booking #: <[bkno  ]> Name <[bclient             ]>

.....................................................
  Sup-   Ship           Depart
  plier  Name   Dest      Date   Description
  ------ ------ -------- -------- -----------------
 [bsup  |bship |bdest   |bdepart |bdesc            ]

=====================================================
}
END
```

3.5 SCREEN USEREXITS

Overview

In this topic we will describe at what point in the screen form run-time activity each userexit would pass control to its C function parameter. Again the form of the userexit clause is:

userexit (*funcname*)

SCREEN userexits include one group for the header portion of the screen form and a parallel group for the array portion of the form. The name of both the header and array userexits are the same, except the array versions are prefixed with an **a**. For example, the **afterrow** userexit in the array is **aafterrow**. Unless otherwise noted, an array userexit works in a way similar to its corresponding header userexit.

The return values from *funcname* that are recognized by SCREENFLEX are listed for each userexit.

BEFORESECTION

The **beforesection** userexit is invoked upon starting the screen section. There is no array version of this userexit since it is only executed once.

Return values:

0 Only value to return.

AFTERSECTION

The **aftersection** userexit is invoked upon exiting the screen section. There is no array version of this userexit since it is only executed once.

Return values:

0 Only value to return.

BEFORESUBSECTION

The **beforesubsection** userexit is invoked immediately upon entering the header portion of the screen form. It is also invoked immediately upon returning to the header portion of the form, having exited the array portion. It is also invoked when changing modes or restarting the header portion. The **abeforesubsection** userexit is invoked immediately upon entering the array portion, if any, of the screen form.

Return values:

0 Only value to return.

AFTERSUBSECTION

The **aftersubsection** userexit is invoked as a last step before leaving the header portion of the screen form, whether that be an exit from the form altogether or an exit to the array portion.

Return values:

0 Only value to return.

The **aaftersubsection** userexit is invoked as a last step before leaving the array portion of the screen form.

Return values:

<0 Block exit from the array.

0 Exit from the array normally.

BEFOREROW

The **beforerow** userexit is invoked after the screen header fields have been displayed but before you begin cursor addressing them.

Return values:

0 Only value to return.

The **abeforerow** userexit is invoked after a screen array row has been displayed but just before you cursor address it.

Return values:

<0 Exit the array portion of the screen form.
0 Take no special action.

ROW

The **row** userexit is not implemented at this time.

AFTERROW

The **afterrow** userexit is not implemented at this time.

BEFOREDISPLAY

The **beforedisplay** userexit is invoked right before the screen form fields are displayed. Any assignments that you make to the screen field variables at this time will be the values in the subsequently displayed fields.

Return values:

0 Only value to return.

BEFORESAVE

The **beforesave** userexit is invoked before the screen data is saved to the database. The save operation occurs when the user presses the **SAVE** key and at least one screen field has been modified. The screen data will NOT have been mapped to the Data base buffer when this userexit is executed.

Return values:

<0 Do not save data. Re-address the field from which the **SAVE** key was pressed. Also, restore the data base buffer to what it was before entering the beforesave() userexit.
0 Proceed with the save procedure.

SAVE

The **save** userexit is invoked in place of the default save procedure of the screen form. **asave** When save() userexit is called, the screen data will have been mapped to the data base fields.

Return values:

<0 Do not invoke **aftersave** userexit and re-address the field from which the **SAVE** key was pressed. Also, restore the data base buffer to what it was before entering the beforesave() userexit.
0 The save procedure was successful. Continue on to any **aftersave** userexit.
1 Use the Default save routine for the screen data.

AFTERSAVE

The **aftersave** userexit is invoked after the screen data is saved to the database. The save operation occurs when the user presses the **SAVE** key and at least one screen field has been modified.

Return values:

<0	Re-addresses the field from which the **SAVE** key was pressed. Also, restore the data base buffer to what it was before entering the beforesave() userexit and reverse the update done by the default save routine.

BEFOREDELETE

The **beforedelete** userexit is invoked when the **DEL** function key has been pressed but before the associated database record is deleted.

Return values:

<0	Do not delete record. Re-address the field from which the **DEL** function key was pressed.
0	Proceed with the delete procedure.

DELETE

The **delete** userexit is invoked in place of the default database record delete procedure of the screen form.

Return values:

<0	Do not invoke **afterdelete** userexit and re-address the field from which the **DEL** function key was pressed.
0	The delete procedure was successful. Continue on to any **afterdelete** userexit.

AFTERDELETE

The **afterdelete** userexit is invoked immediately after the default delete procedure or a successful **delete** userexit.

Return values:

<0	Re-addresses the field from which the **DEL** function key was pressed.

Note that there is a slight complication when using the array version of the *delete* userexits (i.e. **abeforedelete**, **adelete**, and **aafterdelete**). These userexits may be called in ADDMODE as well as CHANGEMODE. To determine which of these modes (ADDMODE vs. CHANGEMODE) is active at the time these *delete* userexits are called perform the following test.

```
if (flexmode == ADDMODE)

if (flexmode == CHANGEMODE)
```

WHEREFUNC

The **wherefunc** userexit is invoked immediately after a table record is read via any of the function keys PREV, NEXT, LAST, FRST. It enables the programmer to selectively view table records from the screen. By returning a -1 from this routine you can reject the record causing the system to behave as if the record did not exist in the table. userexit.

Return values:

-2	End of range reached
-1	Reject record
0	Accept Record

Action Key Userexits

An action key is a keyboard key that invokes some operation on the screen form. Action keys include the function keys, escape key, arrow keys, etc. A userexit may be invoked immediately before the default procedure of every action key. The format for the Action key userexit is as follows:

userexit (*funcname, labelname*)

The *userexit* is the userexit name for the action key (see list below). *Funcname* is the name of the C function to be

called upon pressing the action key. *Labelname* is optional and will overide the action key's default label name displayed at the bottom of the screen.

The return value used from an action key userexit will allow you to control which logic will occur following the userexit. The possible return values and their effect are discussed below.

Return values:

<=1

Do not do the default procedure for the action key. Re-addresses the field from which the action key was pressed.

R_BREAK

Exit from the screen form if in the header portion. Return to header portion if in the array portion.

R_DEFAULT

Proceed with the default operation for the action key.

R_MREPAINT

Repaints and restores the screen to pre_userexit state from memory.

R_DREPAINT

Repaints and restores the screen to pre_userexit state from disk.

These are the userexit names and their label name:

Userexit name	**Keyboard label**
enterkey	RETURN
escapekey	**ESC** (**ESC ESC** on UNIX or XENIX)
backtabkey	**Back Space**
uparrowkey	↑
downarrowkey	↓

Userexit name	**Function key label**
savekey	**DONE** or **SAVE**
jumpkey	**JUMP**
modekey	**ADD** or **CHG**
searchkey	**SRCH**
prevkey	**PREV**
nextkey	**NEXT**
firstkey	**FRST**
lastkey	**LAST**
printkey	**PRNT**
delkey	**DEL**
zoomkey	**ZOOM**
user1key	**USR1**
user2key	**USR2**
helpkey	**HELP**

User1key and **user2key** have no default procedure. They are fully userexit driven.

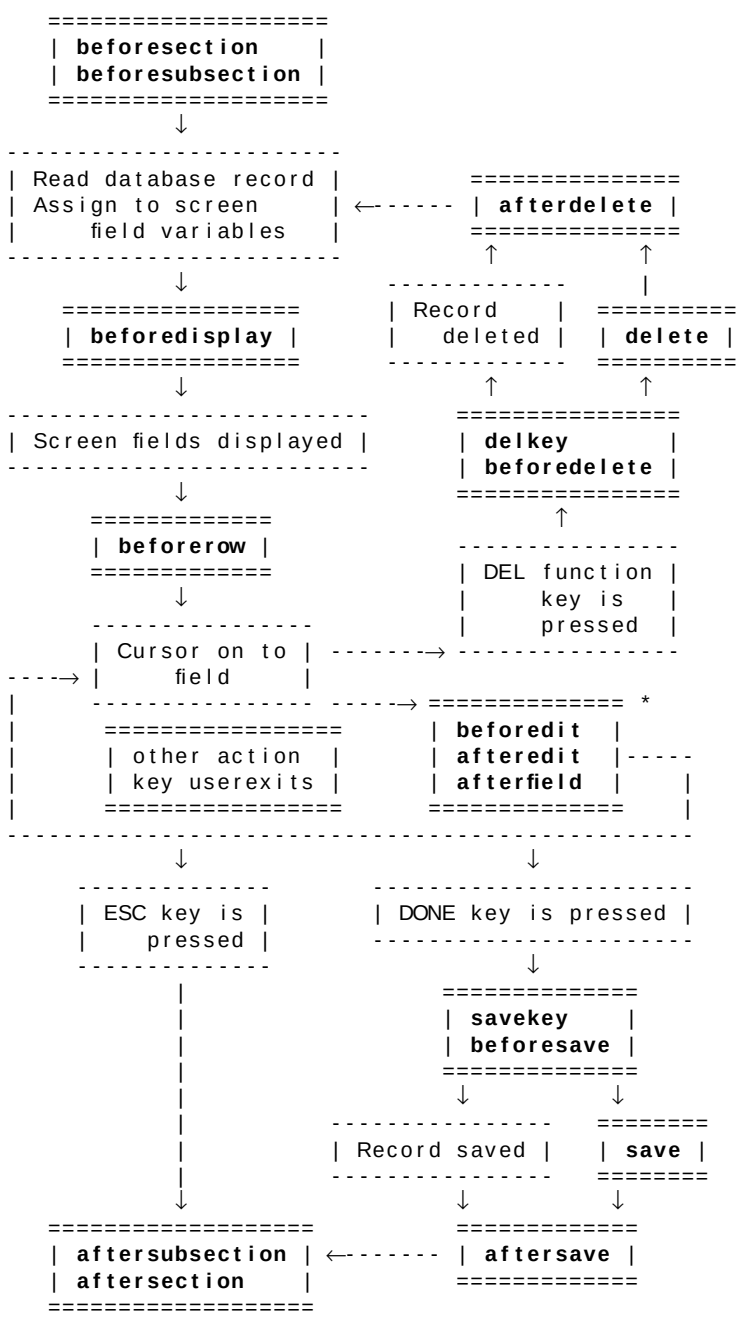
The **zoomkey** userexit will override the **zoomscreen** parameter.

The **helpkey** userexit will override the **helpscreen** parameter.

Note that pressing any of the action keys while addressing a screen form field will also execute the **enterkey** userexit.

Flow Chart

The following is a flow chart to illustrate when each userexit (emboldened) is invoked during the screen form activity:



* The field userexits are **beforedit**, **afteredit**, and **afterfield**. These are described in the next chapter, FIELD ATTRIBUTES.

3.6 ATTRIBUTES SECTION

Overview

The **ATTRIBUTES** section defines the mapping of table fields to screen form fields and specifies the attributes for each field.

Syntax

```
ATTRIBUTES
    [ fieldtag [ ( alttag ) ] = {    table . field
                                   | displayonly type dtype
                                 }   [ , attributes ] ;
    ]
                    .
                    .
    [ REPEAT ( n )
        { fieldtag [ ( alttag ) ] = {    table . field
                                       | displayonly type dtype
                                     }   [ , attributes ] ;
        }
                        .
                        .
      ENDREPEAT
    ]
END
```

Description

ATTRIBUTES	is a required keyword.
fieldtag	is a field label or name from a field of the **SCREEN** section.
alttag	is an optional alternate field label or name.
( )	is the required parenthesis around the optional *alttag* and around the *n* parameter of the optional **REPEAT** clause.
=	is required punctuation after the field tag name specification.
table	is a database table name.
.	is required punctuation between *table* and *field*.
field	is the database field to which *fieldtag* maps.
displayonly	is a keyword used in place of *table*.*field*.
type	is a keyword required with the displayonly keyword.
dtype	is the field's data type specification required with a **displayonly** field.
attributes	are the optional attributes for a field.
,	is the comma that must separate the *attributes*.
;	is the required punctuation at the end of the field specification.
REPEAT	is an optional keyword specifying the beginning of a screen array definition block.
n	is a number parameter of REPEAT. It specifies the number of a screen array rows to display in the form.
ENDREPEAT	is a keyword to mark the end of the REPEAT block, if there is one.
END	is a required keyword to mark the end of the **ATTRIBUTES** section.

Notes

- The **ATTRIBUTES** section must have one *fieldtag* specification for each field in the **SCREEN** section. The order in which *fieldtag*s appear determines the order the cursor will advance through the corresponding fields of the screen form.
- The use of *alttag* is recommended where the *fieldtag* is too short a name, such as a single character, to convey much meaning. Any other reference to this field in the **ATTRIBUTES** section or in any **INSTRUCTIONS** section will used the *alttag*.
- Those fields of the header portion of the screen form that have a *table*.*field* of the primary table of the **SELECT** section will automatically display the contents of selected database records for those screen form fields. Several function keys will select records during run-time: **FRST**, **LAST**, **PREV**, and **NEXT**. When adding or modifying screen form records, the same primary table related fields will be effected.
- Those fields of the array portion of the screen form that have a *table*.*field* of the joined table of the **SELECT** section will automatically display the contents of selected database records for those screen form fields.
- For *dtype* see Chapter 8, DATABASE DATA TYPES.
- The *fieldtag*s within the **REPEAT**/**ENDREPEAT** block represent all the fields of a single row of the screen form array. A single screen form array row may occupy more than one line of the video display.
- Where there is a screen form array, there may be header fields following the array portion of the form.
- An array-only screen form will have no *fieldtag*s outside of the **REPEAT**/**ENDREPEAT** block.
- There are numerous possible *attributes*. The next chapter, FIELD ATTRIBUTES, will be dedicated to this subject.
- The special screen *fieldtag* **modemsg** reserved for displaying the current input mode is define in the ATTRIBUTES section as follows:

```
modemsg = displayonly type character, noupdate, noentry, reverse, retain;
```

Example 1

The following example is a complete screen form source listing, to demonstrate how all the sections interrelate:

```
TABLES
    bkmaster
    bkdetail
    tbven
END

SELECT
    bkmaster( bkno)
        EXTRACTALL
    bkdetail( bkno)  bkmaster( bkno)
        EXTRACTALL
END

SCREEN  bkinput
{
               BOOKING ENTRY SCREEN
=======================================================
Booking #: <[bkno  ]> Name <[bclient             ]>
.......................................................
  Sup-   Ship
  plier  Name   Description
  ------ ------ -----------------------------------
 [bsup  |bship |bdesc                               ]


=======================================================
}
END

ATTRIBUTES
    bkno = bkmaster.bkno, right;
    bclient = bkmaster.name, upshift, required;
    REPEAT( 3)
        bsup = bkdetail.supplier, required, upshift,
               lookup( tbven.tbvenkey,
                       bkinput.bdesc = tbven.desc);
        bship = bkdetail.ship, upshift, left;
        bdesc = displayonly type char;
    ENDREPEAT
END
```

Example 2

This is a source listing for an array-only screen:

```
TABLES
    bkdetail
    tbven
END

SELECT
    bkdetail( bkno)
        EXTRACTALL
END

SCREEN  bkinput
{
               BOOKING ENTRY SCREEN
=======================================================
  Sup-   Ship
  plier  Name   Description
  ------ ------ -----------------------------------
 [bsup  |bship |bdesc                               ]


=======================================================
}
END

ATTRIBUTES
    REPEAT( 3)
        bsup = bkdetail.supplier, required, upshift,
               lookup( tbven.tbvenkey,
                       bkinput.bdesc = tbven.desc);
        bship = bkdetail.ship, upshift, left;
        bdesc = displayonly type char;
    ENDREPEAT
END
```

3.7 RUNNING THE SCREEN FORM

Overview

This topic will describe the command line arguments and the many run-time features of the Infoflex screen form.

Syntax

{ flex | *prog* } *filename* [-f *layout*] [*modes* [*flags*]]

Description

flex	is the default run-time screen form driver.
prog	is the custom run-time driver used when a screen form has an **INSTRUCTIONS** section.
filename	is required and is the base name of the compiled screen form file *filename***.pic**.
-f	is an optional argument prefixing a screen name within screen form file.
layout	is the name of a screen as specified in a **SCREEN** section of the screen form file.
modes	is an optional run-time modes specification for the screen form.
flags	is an optional run-time behavior flags specification for the screen form.

Notes

- You may use the dash character '-' for the *modes* and *flags* arguments to indicate the default value is to be used.
- If there is more than one screen form defined in *filename***.pic** use the **-f** argument to invoke a screen form other than the first one. There must be at least one space between **-f** and *layout*. Without the **-f** argument, the first screen of the screen form file is invoked.
- The *modes* argument is in the form of a sequence of letters, either upper or lower case. Each letter represents a mode:

  A - Add
  C - Change
  D - Delete
  S - Search
  Q - Query
  P - Prompt
  V - View

  The first letter of the sequence represents the initial mode of the screen form. Subsequent letters indicate other modes that are accessible from the form. If the form has an array portion, you may specify a second set of mode letters separated by a dash, **-**. The default modes are **CDSQA-CDSQA**.

  For example:

  flex *filename* ACDQ

  will bring up a screen form in **ADD** mode, with **CHANGE**, **DELETE**, and **QUERY** modes also accessible.
- The *flags* argument is a sequence of **Y**'s and **N**'s, indicating yes or no for the following screen form behaviors.

  1) Repeat the screen form after the **SAVE** key. With a multi-screen form, **JUMPKEY** advances to the next screen form in sequence or loops back from the last screen to the first screen. **ESC** will exit the screen form. Where this flag is set to **N**, pressing the **SAVE** key will exit the program.

2) Unlock the header record upon exiting the screen.
3) Clear the screen form fields when initially entering the screen.
4) Clear the video display upon exiting the screen form.
5) Clear the screen form fields when the **SAVE** key is pressed while in **CHANGE** mode.
6) Clear the screen form fields when the **SAVE** key is pressed while in **ADD** mode.
7) Clear the array region upon exiting the screen form.
8) Use 'F' to automatically position to first record or 'L' to position to last record. This option is only effective if the initial mode is **CHANGE**.

The default sequence of flags is **YYYYNYY**. The default sequence for recursively called screens (i.e. zoomscreens and popups) is **NYYYNYY**.

- If you specify a *flags* argument, you must specify a *modes* argument.
- If a screen form has an **INSTRUCTIONS** section, then the final result of compilation, *prog* is an executable module that is used in place of the **flex** program. See Chapter 2, DEVELOPMENT PROCEDURES, for how to build your custom *prog*.

Example 1

```
flex  bkinput  ACD  NYYYYY
```

Example 2

In this example, **custscr.flx** has an **INSTRUCTIONS** section which results in **custscr** replacing **flex**:

```
custscr  tbemp  -f  commiss
```

CHANGE Mode

Once into an Infoflex screen form. **CHANGE** mode allows a user to select database records and modify the screen form field data.

- In **CHANGE** mode the function key ruler at the bottom of the video display is:

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
SAVE	HELP	----	ADD	----	QRY	PREV	NEXT	FRST	LAST	----	DEL	----	----	----	----

- The *key* fields of the screen form are those fields that are mapped in the source file **ATTRIBUTES** section to the *sortfields* of the database table *mastertable* (see syntax for **SELECT** section under Topic 3.3). Upon entering **CHANGE** mode the cursor will address the first key field of the screen form.
- The **FRST** key will select and display the first record of the *mastertable* ordered by the *sortfields* index.
- **LAST** will select the last record. Entering some value to the key field and pressing **NEXT** will bring up the next record in ordered sequence by the *sortfields* index. For example, assuming that the key field maps to the first field of *sortfields*, which has consecutive values of AA and AC, entering AA or AB and pressing **NEXT** will bring up the record for AC. **PREV** works similarly but selects previous records. Entering a value to the key field and pressing
- With a record displayed, the **SAVE** option will update the database record with any changes the user has made in the screen form.
- For a screen form with an array, **SAVE** from the header portion of the form will display in the array the detail records joined to the header and position the cursor on the first field of the first row, if the initial mode for the array is **CHANGE** mode.
- In the screen array, the database is updated when the user cursors off an array row or presses **SAVE**. The **SAVE** key, in addition to saving the record, will also clear the array display and return the cursor to the key field of the screen form header.
- In a screen array, **NEXT** will display the next page of records, where a page is the number of records that can fit on one screen. **PREV** will display the previous page of records. **FRST** will display the first page, **LAST** the last page. Up and down arrows move up and down rows of the array.
- **DEL** will delete the record. Since **DEL** is such a destructive action, it will prompt the user to confirm the deletion.
- In the screen array, the **DEL** key will delete the current row and pull up rows below to close the gap.
- At any time during **CHANGE** mode, the **ADD** key will switch the form to **ADD** mode.
- In the screen array, pressing the **ADD** key or cursoring off the end of the array will open a new line at the end of the array and invoke **ADD** mode.
- The **HELP** or **HELP** option provides on-line help. The help subsystem is sufficiently elaborate that we will address it in Chapter 11, THE HELP SYSTEM.
- **ESC** will abort and exit the screen (**ESC ESC** on UNIX or XENIX systems). If there are changes that would be lost with this exit, the user is first asked if he wishes to save the record.
- The next chapter will describe individual field behavior as a result of the different field attributes.

ADD Mode

While running an Infoflex screen form, **ADD** mode allows a user to insert a new record into the database.

- In **ADD** mode the function key ruler at the bottom of the video display is:

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
SAVE	HELP	----	CHG	----	----	----	----	----	----	----	----	----	----	----	----

- While on a screen array, if the initial mode is **ADD** mode, a new line will be opened at the end of the array, and the cursor will address the first row of a blank array area.
- The **SAVE** key will attempt to create a *mastertable* record with the screen form fields that map to *mastertable* fields (see use of *mastertable* in the syntax of the **SELECT** section under Topic 3.3).

- The **CHG** key will switch the form to **CHANGE** mode.
- The **HELP** option provides on-line help. The help subsystem is sufficiently elaborate that we will address it in Chapter 11, THE HELP SYSTEM.
- **ESC** will abort and exit the screen (**ESC ESC** on UNIX or XENIX systems). If the user has input data to any fields that would be lost with this exit, he is first asked if he wishes to save the record.
- The next chapter will describe individual field behavior as a result of the different field attributes.

PROMPT Mode

While running an Infoflex screen form, **PROMPT** mode allows a user to enter information that is not related to any database.

- Screen not having a **SELECT** section will be run in **PROMPT** mode.

QUERY Mode

QUERY mode allows the user to locate records on any field or combination of fields on the screen. Values can be specified with wildcard or relational operators. The **User Guide** has a full explanation of the **QUERY** mode features.

VIEW Mode

VIEW mode allows the user to View information from a screen but NOT change it. The screen will behave like **CHANGE** mode but without the editing capabilities.

4. FIELD ATTRIBUTES

A screen form field may have one or more of the following attributes:

AFTEREDIT

Afteredit is a userexit that specifies an **INSTRUCTIONS** section C function to be called immediately after data has been entered to the field for the first time or after data has been modified in a field. The syntax for this attribute is:

```
afteredit( funcname )
```

Returning 0 from *funcname* allows the cursor to advance to the next field. Returning -1 causes the cursor to re-address the current field. Returning -2 causes the cursor to re-address the current field and restore the original value.

AFTERFIELD

Afterfield is a userexit that specifies an **INSTRUCTIONS** section C function to be called upon moving off the field. It is in invoked after any *afteredit* userexit. The syntax for this attribute is:

```
afterfield( funcname )
```

Funcname should always return 0.

AUTOHELP

Autohelp attribute causes the attribute **helpscreen()** or **helpselect()** to be executed if the field is not entered correctly.

AUTONEXT

Autonext causes the cursor to automatically advance to the next field upon entering a character to the last character position of the field. In this case the user is not required to press RETURN to move on to the next field. The next field is determined by the order of the fields in the **ATTRIBUTES** section.

BEFOREDIT

Beforedit is a userexit that specifies an **INSTRUCTIONS** section C function to be called immediately upon cursoring on to the field. The syntax is:

```
beforedit( funcname )
```

Returning 0 from *funcname* allows the cursor to address the field. Returning -1 causes the field to be skipped and advances the cursor to the next field. Returning 4 causes the field to be processed as if the user edited the field. This is useful if you pre-fill the field and you want to make sure all of the editing checks are done.

CENTER

This attribute centers the value within the field. The default is to have the contents of the field left-justified.

CLEAR

This attribute will cause the field to be cleared when the first key is pressed. Numeric fields have this attribute by default.

COMMENTS

This attribute will display a message at the bottom of the video display when the cursor is positioned to the field. The syntax is:

```
comments = "message"
```

where *message* is any character string.

DEFAULT

This attribute assigns a default value to the field. The syntax is:

```
default = value
```

where *value* may be a constant or an SQL field name. If *value* is a constant it must be enclosed in double quotes.

To default to the current date, use the keyword **today** in place of *value*. To default to the current time, use the keyword **time**.

DEFAULTNEXT

This attribute causes the value entered on the previous field to be automatically assigned to this field. Note that the fields should be of the same type.

DEFAULTON,DEFAULTOFF

This attribute can be placed at the screen level or field attribute level. Only fields that have **defaulton** will be saved by the control **D** key feature. The control **D** key saves the current screen values for defaults when in **ADD MODE**.

All report selection screen fields are are assumed to have **defaulton**. All Data entry screens fields are assumed to have **defaultoff**. Using **defaulton** at the screen level sets all fields to **defaulton**.

DOWNSHIFT

Downshift converts uppercase letters in character fields to lowercase.

FORMAT

The **format** attribute is used with DECIMAL, SMALLFLOAT, FLOAT, MONEY, or DATE fields to control the display format. The syntax is:

```
format = "format-string"
```

Format-strings for DECIMAL, SMALLFLOAT, FLOAT, MONEY data types may consist of a pound sign, comma, period, a leading **B** character, or leading **0** character. The pound sign (#) is a placeholder for digits. A comma in the *format-string*

indicates the number is to be formated with commas. The period indicates where the decimal point should appear. For example, the *format-string* "##,###.##" will display two digits to the right of the decimal point. A leading **B** placed in the *format-string* will cause a blank to be displayed whenever the value of the number is zero. A leading **0** placed in the *format-string* will cause leading zeros to be displayed.

For DATE data types there are a number of special formating symbols that may be used. For example, the following format **format="Mmm dd, yyyy Ddd"** will display the date as "Jan 23, 1998 Tue". The formating symbols are described below.

Format	Description
mm	displays a two digit representation of the month
mmm	displays a three letter abbreviation of the month
dd	displays a two digit representation of the day
ddd	displays a three letter abbreviation of the day
yy	displays a two digit representation of the year
yyyy	displays a four digit representation of the year

FORMATFIELD

Formatfield attribute allows you to specify a **C** function to format the screen or report field. The syntax for this attribute is:

```
formatfield( funcname )
```

Unlike other userexits, the **formatfield** userexit passes arguments to your userexit C function.

```
INSTRUCTIONS

funcname( fieldbuffer, displaybuffer, length, mode)
char *fieldbuffer;
char *displaybuffer;
int length;
int mode;
{   /* formats a social security number  with dashes */
   if (mode == 0) { /* Move data from field buffer to display buffer */
       memcpy( displaybuffer, fieldbuffer, 3);
       displaybuffer[3] = '-';
       memcpy( &displaybuffer[4], &fieldbuffer[3], 2);
       displaybuffer[6] = '-';
       memcpy( &displaybuffer[7], &fieldbuffer[5], 6);
       displaybuffer[13] = ' ';
       }

   if (mode == 1) { /* Move data from display buffer to field buffer */
       memcpy( fieldbuffer, displaybuffer, 3);
       memcpy( &fieldbuffer[3], &displaybuffer[4], 2);
       memcpy( &fieldbuffer[5], &displaybuffer[7], 6);
       memcpy( &fieldbuffer[11], "  ", 2);
       fieldbuffer[13] = ' ';
       }
    return(0);
}
END
```

The **formatfield** function arguments are defined as follows:

fieldbuffer

This argument is a pointer to the internal "C" representation of of the field. This buffer will be passed in when the *mode* argument is 0.

displaybuffer

This argument is a pointer to the displayable representation of the field. This buffer will be passed in when the *mode* argument is 1.

length

This integer argument is passed in and represents the length in bytes of *displaybuffer*.

mode

This integer argument is passed in and indicates which direction to convert. When the **mode** is 0, the **formatfield** function should convert the contents of *fieldbuffer* to *displaybuffer*. A **mode** of 1 will be just the reverse.

Returning 0 from *funcname* allows the cursor to advance to the next field. Returning -1 causes the cursor to re-address the current field.

HELPKEY

Helpkey is a userexit that specifies an **INSTRUCTIONS** section C function to be called upon pressing the **HELP** function key. The syntax is:

```
helpkey( funcname)
```

The **helpkey** userexit specified in the **ATTRIBUTES** section will override one specified in the **SCREEN** section. Upon returning from a **helpkey**, the screen will automatically refresh.

HELPSCREEN

Helpscreen calls up another screen upon pressing the **HELP** function key. The syntax is:

```
helpscreen( "scrcommand")
```

Scrcommand is a command line for running a screen that will be helped to. The **helpscreen** command line begins with the *layout* parameter as described in Topic 3.7. The screen specified in *Scrcommand* must exist in the same **.flx** source file as the calling screen. A **helpscreen** specified in the **ATTRIBUTES** section will override one specified in the **SCREEN** section. Upon exiting from a **helpscreen**, the calling screen will automatically redisplay.

HELPSELECT

The **helpselect** attribute is used to provide a popup list of valid codes from another table. The user is able to search and select a valid code while on the popup list. Usually this attribute is used in conjunction with the **lookup** attribute.

The syntax is:

```
helpselect( screenname)
```

Screenname is the name of the Infoflex screen that will display the popup list. The **User Guide** contains an explanation and example on how to use **helpselect**.

INCLUDE

Include defines acceptable values for the field in terms of ranges and lists. The user will not be allowed to cursor off the field until an acceptable value is entered. The syntax is:

```
include( valuelist )
```

where *valuelist* is either a list of individual values:

```
value1 , value2 , value3 , . . .
```

or a range of values:

```
value1 TO value2
```

or a combination of both separated by commas.

LEFT

This attribute left justifies the value in the field. This is the default.

LINENO

The **lineno** attribute is used to automatically assign sequential numbers from a **header** screen field to an **array** screen field. This attribute is specified on the array field holding the sequential number and has the following syntax:

```
lineno( screenname.tagname )
```

The *screenname.tagname* parameter refers to a *tagname* in the **header** portion of the screen that holds the last assigned sequential number. The sequential number is defined as an **integer** in the database.

Below is an example of **lineno's** usage:

```
ATTRIBUTES
   invno = pomaster.invno;
   mlinenum = pomaster.nextlineno,
              noentry, noupdate, nodisplay;
REPEAT(6)
   linenum     = podetail.lineno,
                 noentry, noupdate, lineno( poscreen.mlinenum);
   invenno = podetail.invenno,
END
END
```

LOOKUP

Lookup searches for the field value in the specified database table. The syntax is:

```
lookup( table . key , destfield1 = sourcefield1 ,
                      destfield2 = sourcefield2 ,
                      . . . )
```

*Table*key is the database index name used to search by. The index receives its value from the screen field for which **lookup** is an attribute. The database field making up the index must be identical in type and length to the screen field. The *destfield*s and the *sourcefield*s each have the form *table*.*field* or *screen*.*field*. Typically, a *destfield* is a **displayonly** field of the screen form that receives its value from *sourcefield*, typically a field of the table record that is retrieved in the look-up.

NOCLEAR

Noclear turns off **clear** option for field.

NODISPLAY

The **nodisplay** attribute is used when you wish to store information in a screen field but do not wish it displayed in the screen form.

Non-displayable fields may be below line 20 of the video display and must have the attributes: **nodisplay**, **noentry**, **noupdate**.

NOENTRY

Noentry prevents data entry in a field while in **ADD** mode.

NOUPDATE

Noupdate prevents data entry in a field while in **CHANGE** mode.

PHONE

This attribute will make sure a valid phone number is entered (7 or 10 digits) and then format the number appropriately. Fields using the **phone** attribute must be character type.

REQUIRED

This attribute requires data to be entered into the field. The user will not be allowed to cursor off the field as long as it is blank.

RETAIN

Retain prevents the screen field from being cleared.

RIGHT

This attribute right justifies the value in the field. The default is to have the contents of the field left-justified.

REVERSE

Reverse causes the field to be displayed in reverse video.

SEARCHBY

The **searchby** attribute allows you to specify alternate search indices for retrieving records from a screen form.

- The syntax for this attribute is:

```
searchby( table.index )
```

- *Index* is the name of an index of *table*. The associated field must be all or part of the index.
- At run-time when the user presses the **SRCH** key, he enters **SEARCH** mode, and those fields with the **searchby** attribute will be underlined, and only those fields can be cursor addressed.
- When in **SEARCH** mode the **FIND**, **PREV**, **NEXT**, **FRST**, and **LAST** keys are available and make their record selections base on the index specified by the **searchby** parameter. For the general description of how these keys work, see the Subtopic, CHANGE Mode, under Topic 3.7.
- **EXIT** will exit **SEARCH** mode and return to **CHANGE** mode with the selected record.

SEQUENCE

The **sequence** attribute causes **ADD** mode to insert array records between existing records rather than at the end. This attribute is useful when inputing notes where the user will want to insert a note line between previously entered lines.

To implement this feature, an **integer** field must be defined in the array's table. This field will be used by the program to maintain the order of the array. This **integer** field must also be the part of the array's index (see **SELECT**) that immediately follows the joined portion. Lastly, the **sequence** attribute must be placed on the screen field corresponding to this special **integer** field.

Below is an example of the **sequence** attribute in use:

```
SELECT
   pomord( invno)
       EXTRACTALL
   pomnote(invno, seqno) pomord( invno)
       EXTRACTALL
END
SCREEN  ponote popup box window( 6, 15)
{
         Notes       [invno     ]
[seqno  |note                                          ]



}
END
ATTRIBUTES
  invno  = pomord.invno;
REPEAT(5)
  seqno   = pomnote.seqno, sequence,
            noupdate, noentry, nodisplay;
  note    = pomnote.note;
ENDREPEAT
END
```

There are a couple of pre-programmed *userexits* that may prove useful when using the **sequence** attribute. These *userexits* are **fxreseq** and **fxseqdel**. These are specified as Function key *userexits*:

```
auser1key( fxreseq)  adelkey( fxseqdel)
```

Fxreseq will resequence the array. This function is rarely needed and if so will be requested by the insert routine. **Fxseqdel** will prompt the user for a range of array records to delete. The range is specified in terms of the special **integer** field used for ordering the array.

SETCOL

The **setcol** attribute is used to alter the col position of a screen field when the screen is pre-processed (*fxpp*). This is usually done to overlap fields or move fields to a position outside the boundary of the painted screen. The syntax of this attribute is as follows:

```
setcol( colnum )
```

The *colnum* parameter is is added to the column calculated by the pre-processor. *Colnum* may be negative.

SETROW

The **setrow** attribute is used to alter the row position of a screen field when the screen is pre-processed (*fxpp*). This is usually done to overlap fields or move fields to a position outside the boundary of the painted screen. The syntax of this attribute is as follows:

```
setrow( rownum )
```

The *rownum* parameter is is added to the row calculated by the pre-processor. *Rownum* may be negative.

TOTAL

This attribute will display a total for an screen array field into a header screen field. The syntax is:

```
total( screen . field )
```

TRUNCATE

Same as **clear** attribute keyword.

ULOOKUP

ULookup attribute allows you to specify a **C** function to lookup the screen field. The syntax for this attribute is:

```
ulookup( funcname )
```

UPSHIFT

Upshift converts lowercase letters in character fields to uppercase.

ZOOMKEY

Zoomkey is a userexit that specifies an **INSTRUCTIONS** section C function to be called upon pressing the **ZOOM** function key. The syntax is:

```
zoomkey( funcname )
```

The **zoomkey** userexit specified in the **ATTRIBUTES** section will override one specified in the **SCREEN** section. Upon returning from a **zoomkey**, the screen will automatically refresh.

ZOOMSCREEN

Zoomscreen calls up another screen upon pressing the **ZOOM** function key. The syntax is:

```
zoomscreen( "scrcommand")
```

Scrcommand is a command line for running a screen that will be zoomed to. The **zoomscreen** command line begins with the *layout* parameter as described in Topic 3.7. The screen specified in *Scrcommand* must exist in the same **.flx** source file as the calling screen. A **zoomscreen** specified in the **ATTRIBUTES** section will override one specified in the **SCREEN** section. Upon exiting from a **zoomscreen**, the calling screen will automatically redisplay.

5. REPORTFLEX

REPORTFLEX is the special language of Infoflex report writing.

5.1 SECTION ORGANIZATION

A **REPORTFLEX** source file is an ordinary text file of the form *filename*.**flx**. Its content is subdivided into **sections**. The organization of these **sections** has certain ordering rules.

TABLES Section

The **TABLES** section is a required section and must be the first section of the report source file. Topic 5.2 contains a full description of the REPORTFLEX **TABLES** section.

SCREEN Section

The **SCREEN** section immediately follows the **TABLES** section. It is not a required section of report source file. Topic 5.3 contains a full description of the REPORTFLEX **SCREEN** section.

ATTRIBUTES Section for SCREEN

The **ATTRIBUTES** section for the SCREEN section immediately follows the **SCREEN** section. This section is required if there is a **SCREEN** section, otherwise it is not used. Topic 5.4 contains a full description of the REPORTFLEX **ATTRIBUTES** section.

SELECT Section

The **SELECT** section is required and immediately follows the **ATTRIBUTES** section for the **SCREEN** section. If there is no **SCREEN** section, then the **SELECT** section immediately follows the **TABLES** section. Topic 5.5 contains a full description of the REPORTFLEX **SELECT** section.

There can be more than one **SELECT** section. A **SELECT** section can create a temporary database table that a subsequent **SELECT** section may access. With multiple **SELECT** sections, the sections are arranged consecutively.

REPORT Section

The **REPORT** section is required and immediately follows the **SELECT** section. Topic 5.6 contains a full description of the REPORTFLEX **REPORT** section.

ATTRIBUTES Section for REPORT

The **ATTRIBUTES** section for the REPORT section is required and immediately follows the **REPORT** section. Topic 5.7 contains a full description of the REPORTFLEX **ATTRIBUTES** section.

INSTRUCTIONS Section

The **INSTRUCTIONS** section is an optional last section of the report source file. It contains your C language functions called by any userexits of the report. Chapter 7 contains a full description of the **INSTRUCTIONS** section.

5.2 TABLES SECTION

Overview

The **TABLES** section lists the names of any tables referenced in the report source file.

Syntax

```
TABLES
      tablename    [ tableparms ]
            .
            .
            .
END
```

Description

TABLES	is a required keyword.
tablename	is a database table name.
tableparms	are optional and may be any one of the following: **open**, **read**, **alias** *aliasname*. **Open** parameter will open the file upon starting the program. **Read** parameter will open the file, read the first record, and close the file upon starting the program (usefile for control files). The **alias** *aliasname* parameters will allow you to open the file under a different name.
END	is a required keyword.

Notes

- All the rules for the **TABLES** section of the screen form definition file, Topic 3.2, also apply to the reports.

5.3 SCREEN SECTION

Overview

The **SCREEN** section defines the layout of a report prompting screen form. This form allows the user to specify certain variable parameters of the report.

Syntax

```
SCREEN   select
           [ WINDOW( row , col  [ , height , width ) ]  ]
           [ BOX  |  FRAME  ]   [ POPUP  ]
           [ userexit ( funcname ) ]  .  .  .
{
     literals   [fieldtag          ]   .  .  .
                .
                .
                .
}
END
```

Description

SCREEN	is a required keyword.
select	is the required **SCREEN** name.
WINDOW	is an optional keyword. The **WINDOW** clause specifies alternate video display coordinates where the screen layout will be placed at run-time.
row	Where there is a **WINDOW** clause, this is the top row coordinate. The first row is row 0.
col	Where there is a **WINDOW** clause, this is the leftmost column coordinate. The first column is column 0.
height	Where there is a **WINDOW** clause, this is the optional height or number of rows.
width	Where there is a **WINDOW** clause, this is the optional width or number of columns.
BOX	is an optional keyword that specifies that the screen region defined by **WINDOW** will have a single line border.
FRAME	is an optional keyword that specifies that the screen region defined by **WINDOW** will have a single line border with an additional line separating a title region at the top of the **WINDOW**.
POPUP	is an optional keyword that specifies that the screen will overlay any pre-existing screen.
userexit	is optional and is a userexit name.
funcname	is required with a userexit clause and is the name of a C function that you have written in the screen form's **INSTRUCTIONS** section. *Userexit* will pass program control to this function.
{ }	are required brackets and enclose the screen layout. Each bracket must be on a line by itself.
literals	is that part of the screen layout that will display at run-time exactly as it is represented in the layout.
[ ]	are brackets actually used in the screen layout definition and define the position and length of a given field.
fieldtag	is a label or name for an input or display field of the screen form. This is the name that is used to reference the field in the subsequent **ATTRIBUTES** section.
END	is a required keyword.

Notes

- Without the *height* parameter for the **WINDOW** clause, the height of the window will extend from *row* to the bottom of the screen display. Without the *width* parameter for the **WINDOW** clause, the width of the window will extend from *col* to the right edge of the screen display. Without the **WINDOW** clause altogether, **WINDOW(0,0)** is assumed.
- Field brackets will not appear in the screen form at run-time.
- You may only have *literals* and displayable fields on the first twenty lines of the video display. At run-time REPORTFLEX reserves the bottom three lines for comments, function key labels, prompts, and messages.
- The **select** screen recognizes special keyword *fieldtags*: **rptdest, rptcopies, rpttitle**, and **rptnoprint**.

  The **rptdest** keyword *fieldtag* name indicates a field that will accept input that specifies the report destination. Four inputs are meaningful:

  S - screen
  P - printer
  D - disk
  A - auxiliary port
  O - standard out

  Entering a **P** by itself will cause printer output to be routed to the default printer. To route output to alternative printers the user would enter the printer's device name suffix after the **P**.

  The auxiliary port is a port leading out of many terminals.

  The **rptcopies** keyword *fieldtag* name indicates a field that will accept input that specifies the number of report copies to output. 1 to 10 copies can be specified.

  The **rpttitle** keyword *fieldtag* name indicates a field that will accept input that indicates whether or not a title page should be printed for the report. A report title page is simply a copy of the **select** screen itself with the inputs that the user entered. The user entering **Y** to the **rpttitle** field will print the title page. Entering anything else will not.

  The **rptnoprint** *fieldtag* will allow you to selectively print report layouts based on a **Y** or **N** answer. If a **N** is entered only report layouts without the **rptnoprint** parameter are printed. A typical application of the **rptnoprint** feature is to allow the user to select whether report detail is printed.
- Any other fields of the **select** screen may reference any fields of any tables specified in the **TABLES** section. The ATTRIBUTES section for the **select** screen, the next topic, will define what table fields are mapped to what screen fields.
- There can be any number of userexit clauses, and if more than two, they are space or newline separated. The specific userexits available to the **SCREEN** section are taken up in the SCREEN USEREXITS topic in Chapter 3.
- After the user makes his entries to the **select** screen, pressing **DONE** will start the report. **ESC** (**ESC ESC** on UNIX or XENIX) from the **select** screen will exit the report.
- The **SCREEN** section is an optional section of the report definition file. Without this section the user will not be prompted, and one copy of the report will display to the screen. The **SELECT** section, a subsequent topic, will define what records are selected for the report.

Example

```
SCREEN  select
{
             REPORT SELECTION SCREEN
                  Sales by Agent
=========================================================

Report Destination: [d  ]  (S=Screen, P=Printer,
                            D=Disk,   A=Aux)
Report Copies:      [c ]   (1 - 10)
Report Title Page:  [t]    (Y=Yes, N=No)


Agent:  1) [agen]   [bempname    ]
        2) [age2]   [bempnam2    ]
        3) [age3]   [bempnam3    ]

Booking Date Range: [bdate   ]  to  [edate   ]

Supplier: [sup   ]  [supname         ]

Sales Only: [s]  (Y=Yes or leave blank)

=========================================================
}
END
```

5.4 ATTRIBUTES SECTION FOR SCREEN

Overview

The **ATTRIBUTES** section following the report **SCREEN** section defines the mapping of table fields to screen fields and specifies the attributes for each field.

All the rules for the screen form **ATTRIBUTES** section, Topic 3.6 and Chapter 4, also apply to reports except for the features described in the Notes below.

Syntax

```
ATTRIBUTES
    { fieldtag [ ( alttag ) ] = {    table . field
                                   | displayonly type dtype
                                 }   [ , attributes ] ;
    }
                          .
                          .
END
```

Description

ATTRIBUTES	is a required keyword.
fieldtag	is a field label or name from a field of the **SCREEN** section.
alttag	is an optional alternate field label or name.
( )	is the required parenthesis around the optional *alttag*.
=	is required punctuation after the field tag name specification.
table	is a database table name.
.	is required punctuation between *table* and *field*.
field	is the database field to which *fieldtag* maps.
displayonly	is a keyword used in place of *table*.*field*.
type	is a keyword required with the displayonly keyword.
dtype	is the field's data type specification required with a **displayonly** field.
attributes	are the optional attributes for a field.
,	is the comma that must separate the *attributes*.
;	is the required punctuation at the end of the field specification.
END	is a required keyword to mark the end of the **ATTRIBUTES** section.

Notes

- The use of *alttag* is recommended where the *fieldtag* is too short a name, such as a single character, to convey much meaning. Any other reference to this field in the **ATTRIBUTES** section or in any **INSTRUCTIONS** section will used the *alttag*.
- You must specify the **rptdest** keyword *fieldtag* as **displayonly type character**. The appropriate attributes for the **rptdest** field are:

  upshift
  required

REPORTFLEX does not assume these attributes.

- You must specify the **rptcopies** keyword *fieldtag* as **displayonly type smallint**. The appropriate attributes for the **rptcopies** field are:

  required
  include(1 to 10)

  REPORTFLEX does not assume these attributes.

- You must specify the **rpttitle** keyword *fieldtag* as **displayonly type character**. The appropriate attributes for the **rpttitle** field are:

  upshift
  required
  include(Y,N)

  REPORTFLEX does not assume these attributes.

- What database records are selected for a report can be controlled automatically by the data input to **select** screen fields mapped to *table*.*field*s. To activate this feature, you must use this clause

```
WHERE  whereselect
```

  in the **SELECT** section (see the next topic).

  When using this clause the selection logic is determined by the number of screen *fieldtag*s defined for the same *table*.*field*.

  To select database records with a specific value for a field, define a single *fieldtag* to that *table*.*field*. As an example, for the *fieldtag* of **agen** in our **select** screen, we have this definition in our **ATTRIBUTES** section:

```
agen = bkmaster.agent, . . .
```

  When the user inputs a value to the **agen** field and presses **DONE**, only the **bkmaster** record for that value of the **agent** field will be selected for the report.

  To select database records with a range of values for a field, define two *fieldtag*s to the same *table*.*field*. In the case of our agent field, all **bkmaster** records would be selected whose value for **agent** would be greater than or equal to the value input to the first agent screen field and less than or equal to the value input to the second agent screen field. If no value is entered to the first agent field, then the range would begin at the lowest valued agent. If no value is entered to the second agent field, then the range would end at the highest valued agent.

  To select database records with one of any number of specific values for a field, define three or more *fieldtag*s to the same *table*.*field*. If no values were input, then all records would be selected, one value, one record selected, two values, two records selected, up to the number of screen fields mapped to the single table field.

  Often the records selected for a report are based on some code field. In designing the **select** screen for such a report, it would be a good practice to provide look-ups and table-help for such encoded fields. Refer to the **lookup** and **tablehelp** attributes in the Chapter 4, FIELD ATTRIBUTES.

Example

Here is the **ATTRIBUTES** section to go along with the **SCREEN** section example under Topic 5.3:

```
ATTRIBUTES

d = displayonly type char, default = "S",
    upshift, required, include( S, P, A);
c = displayonly type smallint, default = "1",
    required, include( 1 to 10);
t = displayonly type char, default = "N",
    upshift, required, include( Y, N);

agen = bkmaster.agent, upshift,
       lookup( tbemp.tbempkey,
               select.bempname = tbemp.lname),
       tablehelp( "flex tbemp ASDC",
               tbemp.tbempkey, tbemp.code, tbemp.lname);
age2 = bkmaster.agent, upshift,
       lookup( tbemp.tbempkey,
               select.bempnam2 = tbemp.lname),
       tablehelp( "flex tbemp ASDC",
               tbemp.tbempkey, tbemp.code, tbemp.lname);
age3 = bkmaster.agent, upshift,
       lookup( tbemp.tbempkey,
               select.bempnam3 = tbemp.lname),
       tablehelp( "flex tbemp ASDC",
               tbemp.tbempkey, tbemp.code, tbemp.lname);

bempname = displayonly type char, noupdate, noentry;
bempnam2 = displayonly type char, noupdate, noentry;
bempnam3 = displayonly type char, noupdate, noentry;

bdate = bkmaster.bookdate, default = today;
edate = bkmaster.bookdate, default = today;

sup = bkdetail.supplier, upshift,
      lookup( tbven.tbvenkey,
              select.supname = tbven.name),
      tablehelp( "flex tbven ASDC",
              tbven.tbvenkey, tbven.code, tbven.name);

supname = displayonly type char, noupdate, noentry;

s  = tbven.saleflag;

END
```

5.5 SELECT SECTION

Overview

The **SELECT** section defines how records are to be selected from the database. This section will specify the ordering of the records and to join multiple tables.

Syntax

```
SELECT   [ { userexit ( ufuncname ) } . . . ]
         primarytable ( sortfields )
              [    { tempfield [    { TYPE dtype [ (dlength ) ] }
                                 | = sourcetbl . sourcefld
                              ]
                   } . . .
                 | allfields
              ]
              [ WHERE wherefunc( wfuncname ) ]
              { Relation }
     [ { jointable ( joinfields )    prevtable ( prevfields )
              [    { tempfield [    { TYPE dtype [ (dlength ) ] }
                                 | = sourcetbl . sourcefld
                              ]
                   } . . .
                 | allfields
              ]
              [ WHERE wherefunc( wfuncname ) ]
              { Relation }
       } . . .
     ]
     [ WHERE   [    { wherefunc( wfuncname )
                          [ whereselect ]
                    }
                  | whereselect
               ]
     ]
     [ ONINDEX   temptable ( tempindex ) ]
END
```

Description

SELECT	is a required keyword.
userexit	is optional and is a userexit name.
ufuncname	is required with a userexit clause and is the name of a C function that you have written in the screen form's **INSTRUCTIONS** section.
primarytable	is the required primary table from which records will be selected.
sortfields	is the field or fields that comprise an index of *primarytable*. The selected records are ordered by this index. If there are two or more fields, *sortfields* is a comma separated list.
tempfield	is optional and is the name of a field of *temptable* of the optional **ONINDEX** clause.
TYPE	is a keyword used when specifying the data type of *tempfield*.
dtype	is the data type for *tempfield*.
dlength	is an optional data length specification when *tempfield* is a **char** or **decimal** type .
=	is used in place of the **TYPE** clause to map a database table field to *tempfield*.
sourcetbl	is the database table of the field mapped to *tempfield* with =.

. is required punctuation between *sourcetbl* and *sourcefld*.

sourcefld is a field of *sourcetbl*.

allfields is a keyword to bring all fields of a selected table into *temptable* of the optional **ONINDEX** clause.

WHERE is a keyword of the optional **WHERE** clause. This clause is used to limit the records selected for the report.

wherefunc is a keyword and userexit name of the optional **WHERE** clause. It is required in the **WHERE** clause attached to *primarytable* or a specific *jointable*. It is optional in the final or global **WHERE** clause.

wfuncname is a C function in the **INSTRUCTIONS** section. The **wherefunc** userexit passes control to this function.

Relation There are 7 possible *Relational* operators which can be used when joining two tables. They are:

OUTER

selects all records from *prevtable* and the first record that matches from *jointable*.

OUTERALL

selects all records from *prevtable* and all those that match from *jointable*.

SUBSET

selects only records from *prevtable* that match those from *jointable*. Only the first matching record from *jointable* will be selected.

SUBSETALL

selects only records from *prevtable* that match those from *jointable*. All matching records from *jointable* will be selected.

EXTRACT

This is a combination of OUTER and SUBSET. If any of *jointable* fields are used in the WHERE clause then the relation will behave as SUBSET.

EXTRACTALL

This is a combination of OUTERALL and SUBSETALL. If any *jointable* fields are used in the WHERE clause then the relation will behave as SUBSETALL.

REJECT

selects all records from *prevtable* where there is no match in *jointable*.

Note that the *Relation* operator for the *primarytable*, although required, has no meaning at this time.

jointable is an optional table joined to the primary table.

joinfields is the field or fields that comprise the joined index of *jointable*. The selected *jointable* records are ordered by this index. If there are two or more fields, *joinfields* is a comma separated list.

prevtable is *primarytable* or an earlier *jointable*.

prevfields are the fields of *prevtable* that are joined to *joinfields* of *jointable*.

whereselect is a keyword of the optional final or global **WHERE** clause. It specifies that the report select **SCREEN** may limit the records that are selected for the report. The **whereselect** clause may be used by itself in the **WHERE** clause or in conjunction with a **wherefunc** userexit.

select is the keyword parameter of the **whereselect** clause.

ONINDEX is a optional keyword and specifies a temporary table to be created from all the *tempfield*s and/or **allfields**.

temptable is the temporary table of the optional **ONINDEX** clause.

tempindex is the index created for *temptable*. If there are two or more fields comprising the index, then *tempindex* is a comma separated field list.

END is a required keyword delimiting the end of the **SELECT** section.

Notes

- As a record is selected from *prevtable*, the value of its *prevfields* determines the values given to the corresponding *joinfields* in selecting records from *jointable*. The data type of each field of *prevfields* must be the same as the data type of the corresponding *joinfields*.
- Unlike the SCREENFLEX **SELECT** section, there can be any number of *jointable* blocks.
- The **wherefunc** userexit works differently depending on whether it is associated with a particular *jointable* or it is associated with the entire SELECT statement. The **wherefunc** associated with a *jointable* has the following return code possibilities:

  0 Accepts the current *jointable* record for the report.

  -1 Rejects the current *jointable* record. The *jointable* will be read for the next record.

  -2 Causes end-of-range condition for *jointable*. The *prevtable* will be read for the next record.

  The **wherefunc** associated with the entire SELECT statement is located at the bottom of the SELECT statement and is invoked after all tables are read. This **wherefunc** has the following return code possibilities:

  0 Accepts the current set of records for the report.

  -1 Rejects the current set of records. The *primarytable* will be read for the next record.

  -2 Causes end-of-file condition. No more records will be read.

- One of the *Relational* operators (OUTER, OUTERALL, SUBSET, SUBSETALL, EXTRACT, or EXTRACTALL) must be specified for each *jointable*. OUTER, SUBSET, or EXTRACT is used where there is a one-to-one relationship between *jointable* and *prevtable* (as in the case of a table lookup). OUTERALL, SUBSETALL, or EXTRACTALL is used where there is a one-to many relationship between *jointable* and *prevtable*.
- If the **whereselect** clause is used in the global **WHERE** clause, there must be a **select** screen (see Topic 5.3), where the user may specify selection criteria to reduce the scope of the report.
- The optional **ONINDEX** clause is used to create the temporary table *temptable* with the index *tempindex*. This table will be removed from the database at the point that the user leaves the report.
- Where a **SELECT** section has an **ONINDEX** clause, there will usually be another **SELECT** section immediately following. *Primarytable* and *primaryfields* of that **SELECT** section must correspond to *temptable* and *tempindex* of the **ONINDEX** clause.
- The argument to **TYPE** is a data type expression of a form described in the Chapter 8, DATABASE DATA TYPES.
- There are five possible *userexit*s. We will describe at what point in the table selection activity each userexit would pass control to its *ufuncname*, which at this time must all return 0.

  beforesection. This userexit is called once before the selection process begins. You may return a -1 to prevent further processing.

  section. This userexit is called in place of the selection process. This is the **SELECT** section syntax when using the **section** userexit:

  ```
  SELECT  section( ufuncname )
  END
  ```

  When using the **section** userexit, your data may be outputed thru the REPORT section layouts by calling the C function
  rptprint().

  aftersection. This userexit is called once after the selection process has completed.

  beforerow. This userexit is called immediately before each record is read from *primarytable*.

afterrow. This userexit is called immediately after any record is read from *jointable*, and where there is no *jointable*, after a record is read from *primarytable* record. Where there is an **ONINDEX** clause, *afterrow* is called before an **temptable** record is written.

Example

```
SELECT  afterrow( compcoms)
    bkmaster( bookdate, bkno)
            agent
            salamount
            commiss TYPE money( 7,2)
        EXTRACTALL
    bkdetail( bkno)  bkmaster( bkno)
            detsalamt = bkdetail.salamount
        EXTRACTALL
    WHERE whereselect
    ONINDEX passag( agent)
END

SELECT
    passag( agent)
        WHERE wherefunc( lastchk)
        EXTRACTALL
END
```

- The *tempfield*s are **agent**, **salamount**, **commiss**, and **detsalamt**, and those will be the fields of the temporary table **passag**.
- The data for **agent** and **salamont** comes from the **bkmaster** fields of the same names. The data types for these fields in **passag** will be the same as the data types of the corresponding fields in **bkmaster**.
- We link the name **detsalamt** of **passag** to the **salamount** field of **bkdetail** because **passag** already has the field **salamount** from the **bkmaster** field of the same name.
- The **commiss** field is not directly mapped to any selected table field. In our example we use the **afterrow** userexit to derive the data for **commiss**. The C function **compcoms** is called right after a **bkdetail** record is read but before the **passag** record is written. In the **compcoms** function you can assign to the **commiss** field with:

```
$passag.commiss = value;
```

5.6 REPORT SECTION

Overview

The **REPORT** section specifies the layout of the report.

There are four subsections that make up a **REPORT** section: **heading**, **detail**, **total**, and **footing**.

Syntax

```
REPORT
  [ heading[suf] [breakon(indexfield,n)] [parameters] [userexits]
    {
        literals  [fieldtag       ]  . . .
    }
  ] . . .
  [ detail[suf]  [parameters] [userexits]
    {
        literals  [fieldtag       ]  . . .
    }
  ] . . .
  [ total[suf]  [breakon(indexfield,n) ] [parameters] [userexits ]
    {
        literals  [fieldtag       ]  . . .
    }
  ] . . .
  [ footing[suf]  [userexits]
    {
        literals  [fieldtag       ]  . . .
    }
  ] . . .
END
```

Description

REPORT	is a required keyword.
heading	is a required keyword of the **heading** subsection.
detail	is a required keyword of the **detail** subsection.
total	is a required keyword of the **total** subsection.
footing	is a required keyword of the **footing** subsection.
suf	is a suffix for a subsection name after the first of that subsection. It is a value that makes all subsection names unique within a report source file.
breakon	is an optional keyword used to specify a field of the sort index of the report. When the value changes in this field or a higher order field of the index, the associated **heading** or **total** is output.
indexfield	is a field of the sort index of the report. *Indexfield* is the parameter of the **breakon** clause and has the form *table.field*.
parameters	are one or more optional report format specifications. Where there are more than one, they are space or newline separated. *Parameters* will vary for each subsection.
userexits	are one or two optional userexit clauses. Where there are two clauses, they are space or newline separated.
{ }	are required brackets and enclose a subsection layout. Each bracket must be on a line by itself.
literals	is that part of a subsection layout that will display at run-time exactly as it is represented in the layout.
[ ]	are brackets actually used in a subsection layout definition and define the position and length of a given field.

fieldtag is a label or name for a data field of a subsection. This is the name that is used to reference the field in the subsequent **ATTRIBUTES** section and optional **INSTRUCTIONS** section.

END is a required keyword.

Notes

General

- Though each subsection is optional, a report must have at least one subsection of some kind.
- When the same subsection is repeated more than once, a suffix of your choice must be appended to the subsection name to maintain uniqueness within the subsection names.
- A subsection with a **breakon** clause will output its layout each time its **breakon** index field or a higher order field of the index changes value.
- Character fields when used in the report index can be subindexed, so that you may construct **breakon**s on a certain number of initial characters of the field. For example:

```
breakon( bkmaster.agent, 3)
```

  will output its layout when the value of the first three characters of the **bkmaster.agent** field changes.

- Date fields when used in the report index can be subindexed, so that you may construct **breakon**s for different parts of the date. For example:

	Breaks on
`breakon( bkmaster.bookdate, 1)`	year
`breakon( bkmaster.bookdate, 2)`	month
`breakon( bkmaster.bookdate, 3)`	day
`breakon( bkmaster.bookdate, 4)`	week

- A userexit clause has the format:

  userexit (*funcname*)

  There are two userexits available to every subsection. We will describe at what point in the report output activity each userexit would pass control to its *funcname*.

  beforeprint. This userexit is called before each printing of the associated subsection's layout. If *funcname* returns 0, the associated subsection will output its layout. A return value of -1 prevents the output.

  afterprint. This userexit is called after each printing of the associated subsection's layout. *Funcname* should always return 0.

  Beware that when using these userexits for a **total** subsection, the database buffers will contain information for a record beyond the scope of the **total** subsection's **breakon** clause (otherwise the **breakon** condition would never have been triggered).

- To save space used by the field brackets, **[ ]**, you may use the vertical bar **|** to mark the end of one field and the beginning of another.
- Field brackets or field delimiting vertical bars will not appear in the report output.

Heading

- Without the **breakon** clause, a **heading** is output at the top of every page. To have **breakon** headings print at the top of each, you will need to use the **everypage** paramter (see below).

- *Parameters* appropriate to the primary **heading** are:

pagelength(*n*)	number of physical lines per page (default 66). Specifying a page length of 999 will prevent the **newpage** condition (see below) from ever happening.
printlength(*n*)	maximum number of print lines allowed per page (default 56)
printline(*n*)	line number where subsection will begin printing
newpage	break to a new page
everypage	forces a **breakon** heading to print at the top of each new page as well as when the **breakon** occurs.
compress	prints in compressed mode on the output printer
pitch12	outputs 12 characters per inch instead of 10 (printer only, where supported)
nokill	prevents the user from aborting the report durring execution.

Detail

- The **detail** subsection outputs information as each record is read at the lowest level of the relation.
- When there is a parallel one-many relation (the **SELECT** section has a table with more than one EXTRACTALL table joined to it), a **detail** subsection is defined for each one of these parallel relations. As the report is processed, the first parallel relation is completely read and outputted with the first **detail** subsection, then the second parallel relation is read and outputted with the second **detail** subsection, and so on.
- The **printline()** or **printlength()** parameters may be used for this subsection. The **printlength** parameter can be used to limit the number of printed **detail** lines per page.

Total

- All numeric fields are automatically totaled. Fields, such as numeric codes, that you do not want totaled should have the **nototal** attribute in the **ATTRIBUTES** section.
- Without the **breakon** clause, a grand **total** is output at the end of the report.
- The **printline()** parameter may be used for the this subsection.

Footing

- The **footing** subsection outputs at the bottom of each report page.

Example

```
REPORT

heading  breakon( bkmaster.agent) pitch12
{
                SALES ANALYSIS REPORT
==================================================
[bemp]  [bempname       ]
   Book Date Book No Name          Amount
   --------- ------- ------------- --------------
}
heading breakon( bkmaster.bkno) beforeprint( calc)
{
   [bdate   ][bkno  ][bclient      ][bsalam      ]
}

detail  afterprint( wrapup)
{
             [bsup  ][bship |bdest  |dsalam      ]
}

total  breakon( bkmaster.bookdate, 4)
{
  * *   SUBTOTAL by Week   * *     $[dsalam      ]
}
total  breakon( bkmaster.bookdate,2)
{
  * *   SUBTOTAL by Month  * *     $[dsalam      ]
}
total  breakon( bkmaster.agent)
{
  * *   SUBTOTAL by Agent  * *     $[dsalam      ]
}
total
{
  * *   GRAND TOTALS       * *     $[dsalam      ]
}

footing
{
==================================================
                                        PAGE [page]
}

END
```

5.7 ATTRIBUTES SECTION FOR REPORT

Overview

The **ATTRIBUTES** section following the report **REPORT** section defines the mapping of table fields to report fields and specifies the attributes for each field.

Syntax

```
ATTRIBUTES
    { fieldtag [ ( alttag ) ] = {    table . field
                                   | displayonly type dtype
                                 }   [ , attributes ] ;
    }
                          .
                          .
END
```

Description

ATTRIBUTES	is a required keyword.
fieldtag	is a field label or name from a field of the **SCREEN** section.
alttag	is an optional alternate field label or name.
( )	is the required parenthesis around the optional *alttag*.
table	is a database table name.
.	is required punctuation between *table* and *field*.
field	is the database field to which *fieldtag* maps.
displayonly	is a keyword used in place of *table*.*field*.
type	is a keyword required with the displayonly keyword.
dtype	is the field's data type specification required with a **displayonly** field.
attributes	are the optional attributes for a field.
,	is the comma that must separate the *attributes*.
;	is the required punctuation at the end of the field specification.
END	is a required keyword to mark the end of the **ATTRIBUTES** section.

Notes

- The **ATTRIBUTES** section must have one *fieldtag* specification for each field in the **REPORT** section.
- Those fields with *fieldtag*s that are mapped to *table*.*field*s of a record selected in the report's **SELECT** section will automatically output the contents of those selected database fields.
- You may reuse the same *fieldtag* within different subsections of the **REPORT** section, as long as you do not use a *fieldtag* more than once within the same subsection. This eliminates redundant *fieldtag* definitions in the **ATTRIBUTES** section.
- The use of *alttag* is recommended where the *fieldtag* is too short a name, such as a single character, to convey much meaning. Any other reference to this field in the **ATTRIBUTES** section or in any **INSTRUCTIONS** section will used the *alttag*.

- Three **displayonly** fields are frequently used in report **heading**s, the date, the time, and the page number. These fields are defined this way in the **ATTRIBUTES** section:

```
tday = displayonly type date, default=today;
tim  = displayonly type mtime, default=time;
page = displayonly type smallint;
```

  The keyword **today** evaluates to today's date. The keyword **time** evaulates to the current time or the time at which the first **heading** of the report is generated. **Page** is a keyword field tag indicating the field in which the page number will appear.

- For *dtype* see Chapter 8, DATABASE DATA TYPES.
- The report-only attribute **nototal** causes a numeric field of a **total** subsection not to be totaled. At times you may have a numeric field, such as a code, where generating a total for it would be inappropriate.
- Only two other attributes have effect in reports. **Right** will right justify the data in the field. **Format** will allow you to specify the representation of numeric fields (see Chapter 4, FIELD ATTRIBUTES).

Example

```
ATTRIBUTES

bemp     = bkmaster.agent;
bempname = tbemp.lname;

bdate    = bkmaster.bookdate;
bkno     = bkmaster.bkno, right;
bclient  = bkmaster.name;
bsalam   = bkmaster.salamount, format="#,###.##";

bsup     = bkdetail.supplier;
bship    = bkdetail.ship;
bdest    = bkdetail.destination;
dsalam   = bkdetail.salamount, format="#,###.##";

page    = displayonly type smallint;

END
```

5.8 RUNNING THE REPORT FORM

Overview

This topic will describe the command line arguments and the many run-time features of the Infoflex report form.

Syntax

{ flex | *prog* } *filename* [*dest*] [*copies*] [*title*]

Description

flex	is the default run-time report form driver.
prog	is the custom run-time driver used when a report form has an **INSTRUCTIONS** section.
filename	is required and is the base name of the compiled report form file *filename***.pic**.
dest	is an optional argument to specify the report output destination. The default is **S** (Screen) when a **select** screen is specified or **P** (Printer) if no **select** screen is specified. Other destinations available are listed under the REPORTFLEX Screen Section topic. When using destination **P** (Printer) or **D** (Disk) you may specify the name of the printer or disk file by concatinating the name onto the destination argument. For example, 'Drptfile' will route the report to the disk file name 'rptfile'.
copies	is an optional argument to specify the number of copies. The default is **1**.
title	is an optional argument to specify wether to print a title page preceding the report. The title page is a printout of the select screen. The default is **N**.

Notes

- If the report form does not include a **select** screen then the report will run immediately and will be sent to the printer.
- If a report form has an **INSTRUCTIONS** section, then the final result of compilation, *prog* is an executable module that is used in place of the **flex** program. See Chapter 2, DEVELOPMENT PROCEDURES, for how to build your custom *prog*.

Example 1

```
flex  tbemp  P 10 N
```

Example 2

In this example, **tbemp.flx** has an **INSTRUCTIONS** section which results in **tbemp** replacing **flex**:

```
tbemp tbemp P 10 Y
```

6. MENUFLEX

This chapter explains how to build menus .

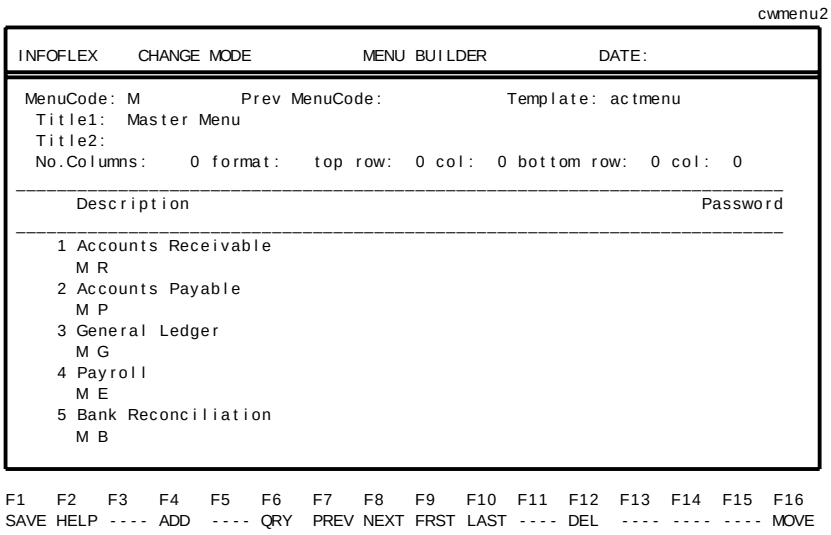
Before developing menus, the **fxmkapp** utility, described under the Application Set-Up topic under Chapter 2, DEVELOPMENT PROCEDURES, must be run in order to build the menu database and install a default run-time menu and the **cwmenu** utility.

6.1 MENU BUILDING

Menu definitions are stored as data in the database. The **cwmenu** utility is an Infoflex screen form that allows you to modify the menu database tables. To build menus, select the menu choice **Design Menus** on the **Development Menu** or enter the following command.

cwmenu cwmenu2

The following entry screen for building menus will appear.



The menu builder screen behaves and adheres to the same rules as for any other Infoflex entry screen.

Below is a description of each screen field.

Menu Code

This is the code name for a menu. We recommend that it be short, such as M for master menu or P for accounts payable menu, though it can be up to 10 alphabetic characters in length. Do not use numbers in the menu code.

Previous Menu Code

This is the code name for the menu that calls the menu you are defining. A main menu, of course, would not have a previous menu.

Template

You may create custom menu formats called templates. The template name is the first parameter of a **MENU** section in **menu.flx** The template name can be up to 18 characters in length. Where the **template** field is left blank, the menu format will be automatically generated at run-time. For a technical discussion about creating templates, please refer to your Infoflex-4GL manual Chapter 6.5.

No. of Columns

A menu can list its options in single or multi-column format. By default, the menu choices will be listed in a single column. To organize menu choices into multiple columns you would specify the number of columns here. Note that a 0 or blank for the number of columns is the same as 1. This value is ignored where a template has been specified.

Title

There can be up to two lines of 76 characters each for the menu title.

Format

There are three format options for a menu. The default, which is no value for format, will display a frame around a menu. Option **N** will not display the frame. Option **O** is for overlayed menus. An overlayed menu will display in a bordered window.

Top Row, Col, Bottm Row, Col

If you have specified a overlay format (previous field is set to 'O') then you will need to provide the coordinates for the overlay. The coordinates are entered as the row and column of the upper left corner and the row and column of bottom right corner.

The array portion of the menu builder screen allows you to define each menu choices. Below is a description of each array field.

Description

The option description or text can be up to 60 characters in length. If you define a menu template with shorter fields or use a two column format, your option text will be truncated.

Password

The **Password** may be 8 characters in length and is used to restict access to a menu choice. A menu choice having a password will require the user to enter the password in order to access the program specified in the **Execution Line** (see below).

Execution Type

At run-time a menu option may invoke another menu, a program, or a function. The execution type is specified with an **M, P, or F**.

Execution Line

For an execution type of **M** the execution line for the option is a menu code. For an execution type of **P** the execution line is a program command, such as:

flex *filename*

For an execution type of **F** the execution line is a function call. In the third case the **menu.pic** file must be invoked by a custom Infoflex application program, that is, a program in which **INSTRUCTIONS** section C language functions are linked (more on this in the Infoflex-4GL manual Chapter 7, THE INSTRUCTIONS SECTION).

Clear Screen Flag

Entering **Y** to this field will cause the screen to clear when the menu option is invoked. Entering **N** or leaving it blank will cause the screen not to be cleared. Note that Infoflex or Accountflex screens will automatically clear without having to set the **Clear Screen Flag** equal to **Y**.

Before Option RETURN

If you enter **Y** to this field, the menu will require the user to enter RETURN before executing the option. Entering **N** to this field or leaving it blank will result in the default behavior of not prompting.

After Option RETURN

If you enter **Y** to this field, the menu will require the user to enter RETURN upon returning to the menu after executing the option. Entering **N** to this field or leaving it blank will result in the default behavior of not prompting.

6.2 RUNNING THE MENU

Assuming the menu code for your main menu is **M**, the user would start the application with

flex menu M

This should then be the entry point of access to all screens, reports, and submenus of your application.

With a menu displayed, arrow keys will move the cursor over the menu options. The user may select an option by positioning the cursor and pressing RETURN or by entering an option number in the option selection field and pressing RETURN.

Pressing **ESC** (**ESC ESC** on UNIX or XENIX systems), **e** or **b** will exit the menu the user is in and return to the previous menu or exit the program altogether if the user is leaving the main menu.

Pressing **!** will allow the user to then type and execute an operating system command while still in the menu.

Pressing the **F3** key will allow the user to chain to any option of any menu of the application. To the chain prompt, the user must enter the menu code immediately followed (no spaces) by the option number. Pressing RETURN will invoke the option.

6.3 MENU TEMPLATE ORGANIZATION

This topic describes the structure of the menu template. A template allows you customize a menu layout in a manner similar to defining screen forms. Place your menu templates in a special file name **menu.flx**. This file is an ordinary text file and its content is subdivided into **sections**. The organization of these **sections** has certain ordering rules.

TABLES Section

The **TABLES** section is a required section and must be the first section of **menu.flx**. Topic 6.4 contains a full description of the MENUFLEX **TABLES** section.

MENU Section

The **MENU** section is an optional section and immediately follows the **TABLES** section. Topic 6.5 contains a full description of the MENUFLEX **MENU** section.

ATTRIBUTES Section

If there is a **MENU** section then there is an **ATTRIBUTES** section immediately following. Topic 6.6 contains a full description of the MENUFLEX **ATTRIBUTES** section.

INSTRUCTIONS Section

The **INSTRUCTIONS** section is an optional last section of **menu.flx**. It contains your C language functions called by any userexits of the menu. Chapter 7 contains a full description of the **INSTRUCTIONS** section.

Multiple Menu Definitions

More than one menu can be defined in **menu.flx**. The entire file would have a single **TABLES** section at the beginning, and any number of the sequences of **MENU**/**ATTRIBUTES** sections. For example, this could be the outline of a screen source file with three menu definitions:

TABLES section

MENU section
ATTRIBUTES section

MENU section
ATTRIBUTES section

MENU section
ATTRIBUTES section

6.4 TABLES SECTION

Overview

The **TABLES** section lists the names of any tables referenced in the menu source file.

Syntax

```
TABLES
        sysfile
        menuhead
        menufield
        menuuser
        menuusty
        menuperm
      [ tablename(s)   ]
END
```

Description

TABLES	is a required keyword.
sysfile	is a required table name in the **TABLES** section.
menuhead	is a required table name.
menufield	is a required table name.
menuuser	is a required table name for menu security.
menuusty	is a required table name for menu security.
menuperm	is a required table name for menu security.
tablename	is one or more optional database table names.
END	is a required keyword.

Notes

- Where there are no custom templates, the **TABLES** section is the only section in **menu.flx**.

6.5 MENU SECTION

Overview

The optional **MENU** section defines a custom layout of a menu. This user-defined menu layout is called a menu template.

Syntax

```
MENU   templatename  [  userexits  ]
{
        literals
     [  opttag                    ]   .  .  .
                    .
                    .
                    .
     [s  ]
}
END
```

Description

MENU	is a required keyword.
templatename	is the required name of the custom menu layout or template. It must represent a unique identifier among all menus within **menu.flx**.
userexits	are one or two optional userexit clauses. Where there are two clauses, they are space or newline separated.
{ }	are required brackets and enclose the menu layout. Each bracket must be on a line by itself.
[ ]	are brackets actually used in the menu layout definition and define the position and length of a given field.
opttag	is a label or name for a menu option field. This is the name that is used to reference the field in the subsequent **ATTRIBUTES** section and optional **INSTRUCTIONS** section.
s	is a keyword tag name for the field from which the user may specify an menu option during run-time.
literals	is any part of the menu layout outside of **[ ]**. This area will display at run-time exactly as it is represented in the layout.

Notes

- The text of a menu option is displayed in a field tagged with *opttag*. The *opttag* name must be unique within the template.
- At run-time the menu option number, the **itemnum** field of the **menufield** record, will display just left of the option field.
- At run-time the value in the system name field of the **sysfile** table will display in the top left corner of the menu. See option 6, Modify Central File, of the Development Menu, Chapter 2.
- At run-time the value in the company name field of the **sysfile** table will display in the top center of the menu.
- The *userexits* clause has the syntax:

  userexit (*funcname*)

  Funcname is the name of a C function that you have written in the menu's **INSTRUCTIONS** section. We will

describe at what point in the menu activity each *userexit* would pass control to its *funcname*. *Userexit* is one of the following:

beforesection. This userexit is called once when the menu is first displayed.

aftersection. This userexit is called at the point that a menu exit key is pressed.

Example

```
MENU bookmenu
{

   Data Entry Programs:         Reports:
   --------------------         --------
    [m1               ]     [m11              ]
    [m2               ]     [m12              ]
    [m3               ]     [m13              ]
    [m4               ]     [m14              ]
    [m5               ]     [m15              ]
    [m6               ]
    [m7               ]     End of the Month:
    [m8               ]     -----------------
    [m9               ]     [m16              ]
                            [m17              ]
      Query On-Line:        [m18              ]
      --------------
    [m10              ]

            Enter Selection > [s ]
}
END
```

6.6 ATTRIBUTES SECTION

Overview

The **ATTRIBUTES** section following the **MENU** section simply maps the menu option fields to the **itemdesc** field of the **menufield** table.

Syntax

```
ATTRIBUTES
     opttag = menufield.itemdesc [ , userexits ];
                    .
                    .
                    .
     s = displayonly type char;
END
```

Description

ATTRIBUTES	is a required keyword.
opttag	is a field label or name from an option field of the **MENU** section.
menufield	is the table associated with every option field.
itemdsesc	is the **menufield** field to which every option field is mapped.
userexits	are one or two optional userexit attributes. Where there are two, they are comma separated.
s	is the required keyword field tag for the option selection field.
displayonly	is a required keyword for the **s** field tag.
type	is a required keyword for the **s** field tag.
char	is a required keyword for the data type of the **s** field.
END	is a required keyword.

Notes

- The ordering of the menu options at run-time is determined by the value of the **itemnum** field of the **menufield** record.
- The two userexit attributes **beforedit** and **afteredit** are available with menu option fields. See Chapter 4, FIELD ATTRIBUTES.

6. MENU SECURITY

This chapter describes the Infoflex menu security system. This chapter will show you how to (1) assign user passwords and (2) control user access to menu choices. Both of these security options may be user specific.

There are 2 steps in setting up menu security. The first step is to define **User Types** and their respective menu permissions. The second step is to define each **User Account** and assign them their **User Type**.

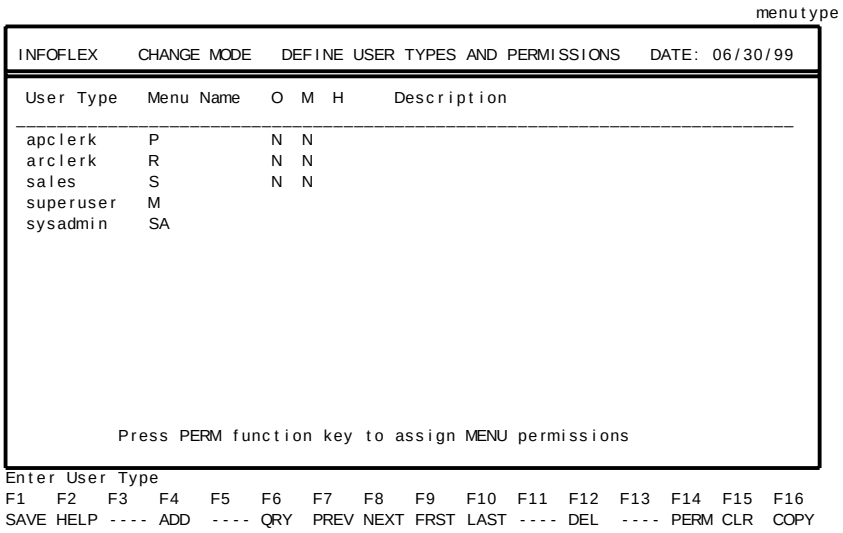
The following sections will describe each of these steps in greater detail.

6.1 Defining User Types

To define **User Types**, select the menu choice **Define User Types** on the **Development Menu** or enter the following command.

cwmenu cwmenu2 -f menuustytable

The screen below will appear.



Below is a description of each field on the above screen.

User Type

Enter a user type. Define a user type for each group of users that will have like permissions.

Menu Name

Enter the menu name the user will start with. The menu name is the unique code assigned to each menu (also called menu code). When running the menus, this code is displayed at the top of each menu within parenthesis. You may press the **HELP** key to select or search from a popup list of valid entries.

O

Enter 'N' to NOT permit access to the Operating System.

M

Enter 'N' to NOT permit menu jumping (F3 key from any MENU).

H

Enter 'N' to NOT permit zoom access from within HELP, or enter 'V" to permit zoom access but in VIEW mode only. Zoom access from HELP allows users to add new entries to the HELP popup list by pressing the **ZOOM** function key.

Description

This is a freeform description field.

To specify menu permissions for a **User Type**, press the **PERM** Function key. Upon pressing the **PERM** key, the Menu Permission screen will appear as shown below.

```
                                                                menuperm1
  MENU PERMISSIONS
|                  Master Menu                    |RMISSIONS   DATE: 06/30/99
 |            (User Type: superuser)               |
 |_________________________________________________ |
_|    1 Accounts Receivable                        |_____________________________
 |    2 Accounts Payable                           |
 |    3 General Ledger                             |
 |    4 Payroll                                    |
 |    5 Bank Reconciliation                        |
 |    6 Inventory                                  |
 |    7 Bill of Materials                          |
 |    8 Sales                                      |
 |    9 Purchase Order                             |
 |   10 Job Cost                                   |
 |   11 System-Wide Control                        |
 |   12 QUEUE                                      |
 ________________________________________________

          Press PERM function key to assign MENU permissions

Enter '-' to deny perm, '=' to not show, and 'V' for View only
F1   F2   F3   F4   F5   F6   F7   F8   F9   F10  F11  F12  F13  F14  F15  F16
SAVE HELP ---- ---- ---- QRY  PREV NEXT FRST LAST ---- ---- ---- ZOOM ---- ----
```

This is a *MULTI-RECORD* screen listing the menu choices found on the **User Type's** starting menu. The column preceeding each menu choice is where you will grant permissions. This column may be one of 4 values.

Value	Description
blank	allow permission to execute the menu choice.
-	deny permision to execute the menu choice.
=	do not show menu choice.
V	permit the menu choice but in view mode only. This option is applicable if the menu choice calls an Infoflex data entry program.

If a menu choice calls another menu, press the **ZOOM** key to bring up its Menu Permission screen. For example, pressing **ZOOM** while cursored on the Accounts Receivable menu choice will bring up the following Menu Permission screen.

```
                                                                menuperm2
  ___MENU PERMISSIONS_____________________________
|     ___MENU PERMISSIONS_____________________________ONS   DATE: 06/30/99
 |     |              Accounts Receivable              |
 |_____|            (User Type: superuser)             |
_|    1|_____________________________________________ |____________________
 |    2|    1 Enter Invoices                          |
 |    3|    2 Print Batches                           |
 |    4|    3 Post Batches                            |
 |    5|    4 Print Journal                           |
 |    6|    5 Enter Adjustments                       |
 |    7|    6 Print Batches                           |
 |    8|    7 Post Batches                            |
 |    9|    8 Print Journal                           |
 |   10|    9 Enter Receipts                          |
 |   11|   10 Print Batches                           |
 |   12|   11 Post Batches                            |
 ______|   12 Print Journal                           |
       ______________________________________________

          Press PERM function key to assign MENU permissions
Enter '-' to deny perm, '=' to not show, and 'V' for View only
F1   F2   F3   F4   F5   F6   F7   F8   F9   F10  F11  F12  F13  F14  F15  F16
SAVE HELP ---- ---- ---- QRY  PREV NEXT FRST LAST ---- ---- ---- ZOOM ---- ----
```

If the menu choice does not call another menu but instead executes a program, the following Menu Permission screen will appear.

```
                                                                menuperm3
  ___MENU PERMISSIONS_____________________________
|     ___MENU PERMISSIONS_____________________________ONS   DATE: 06/30/99
 |     |              Accounts Receivable              |
 |_____|            (User Type: superuser)             |
_|    1|_____________________________________________ |____________________
 |    2|    1 Enter Invoices                          |
 |    3|    2 Print Batches                           |
 |    4|    3 Post Batches                            |
 |    5|    4 Print Journal                           |
 |    6|    5 Enter Adjustments                       |
 |    7|    6 ________________________________________________
 |    8|    7 |                Enter Invoices                 |
 |    9|    8 |            (User Type: superuser)             |
 |   10|    9 |_____________________________________________ |
 |   11|   10 |Permission List:                               |
 |   12|   11 |Overide Command:                               |
 ______|   12 |Default Command: arflex arinv                 |
       _______|                                               |
              ________________________________________________
          Press PERM function key to assign MENU permissions
Enter Special Permission codes (specific to program)
F1   F2   F3   F4   F5   F6   F7   F8   F9   F10  F11  F12  F13  F14  F15  F16
SAVE HELP ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
```

The following is a description of each field on the above screen.

Permission List

This field is for listing permission keywords that will be passed to the program executed by this menu choice. Permission keywords are used to provide program specific security. For example, if you do not want to show the Social Security Number when accessing the employee entry screen you would specify the permission keyword **NOSSN**.

Overide Command

You may specify a command to be executed when this menu choice is selected. This command will overide the default command which is displayed below.

Default Command

This is the command that will be executed when this menu choice is selected unless the **Overide Command** is specified above. This is a view only field.

Besides the **PERM** function key, there are two other important functions keys available when defining **User Types**: **CLR** and **COPY**. The **CLR** function clears previous menu permission settings. THe **COPY** key copies menu permissions from one **User Type** to another. Each of these functions is shown below.

When you press the **CLR** function key, the following screen will appear.

```
                                                                    menuclr

INFOFLEX    CHANGE MODE   DEFINE USER TYPES AND PERMISSIONS   DATE: 06/30/99

 User Type   Menu Name   O  M  H     Description
_____________________________________________________________________________
 apclerk     P           N  N
 arclerk     R           N  N
 sales       S           N  N
 superuser   M

 ____________________________________________________
 |         Clear permissions for User Type:          |
 |                                                   |
 |                                                   |
 |             Press SAVE function key to clear      |
 |                 Press ESCAPEKEY to abort          |
 |                                                   |
 ____________________________________________________

         Press PERM function key to assign MENU permissions

F1   F2   F3   F4   F5   F6   F7   F8   F9   F10  F11  F12  F13  F14  F15  F16
SAVE HELP ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
```

To clear a specific **User Type**, enter the **User Type** in the prompt provided. To clear ALL **User Types** leave the prompt empty.

When you press the **COPY** function key, the following screen will appear.

```
                                                                    menucopy

INFOFLEX    CHANGE MODE   DEFINE USER TYPES AND PERMISSIONS   DATE: 06/30/99

 User Type   Menu Name   O  M  H     Description
_____________________________________________________________________________
 apclerk     P           N  N
 arclerk     R           N  N
 sales       S           N  N
 superuser   M
 sysadmin    ____________________________________________
             |  Copy permissions for User Type: sysadmin |
             |                    to User Type:          |
             |                                           |
             |         Press SAVE function key to copy   |
             |            Press ESCAPEKEY to abort       |
             |                                           |
             ____________________________________________

         Press PERM function key to assign MENU permissions


F1   F2   F3   F4   F5   F6   F7   F8   F9   F10  F11  F12  F13  F14  F15  F16
SAVE HELP ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
```

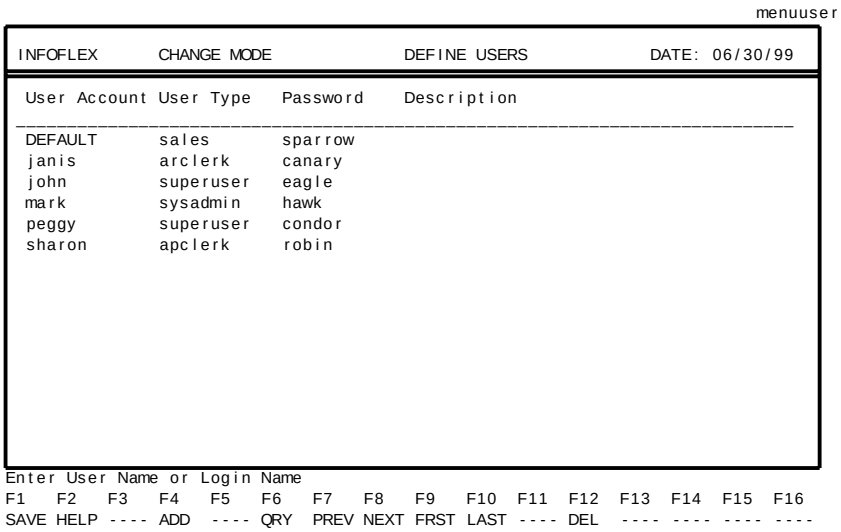
To copy permissions from one **User Type** to another, fill in the *for* and *to* **User Types** on the above screen.

6.2 Defining Users

To assign users their **User Type** permissions, select the menu choice **Define Users** on the **Development Menu** or enter the following command.

cwmenu cwmenu2 -f menuusertable

The following screen will appear.



Below is a description of each field on the above screen.

User Account

Enter a valid user account or sometimes called user login. User accounts are created from within UNIX or NT and are prompted for at login (refer to your operating system's System Administration guide for further information).

Note that there is a special user account called **DEFAULT** which may be optional entered. The **DEFAULT** account will be used for any undefined user accounts accessing the system.

User Type

Enter a valid user type to assign to this user account. You may press the **HELP** key to select or search from a popup list of valid entries.

Password

Enter a password for this user account. This is an optional entry and, if entered, will require the user account enter this password prior to bringing up Infoflex.

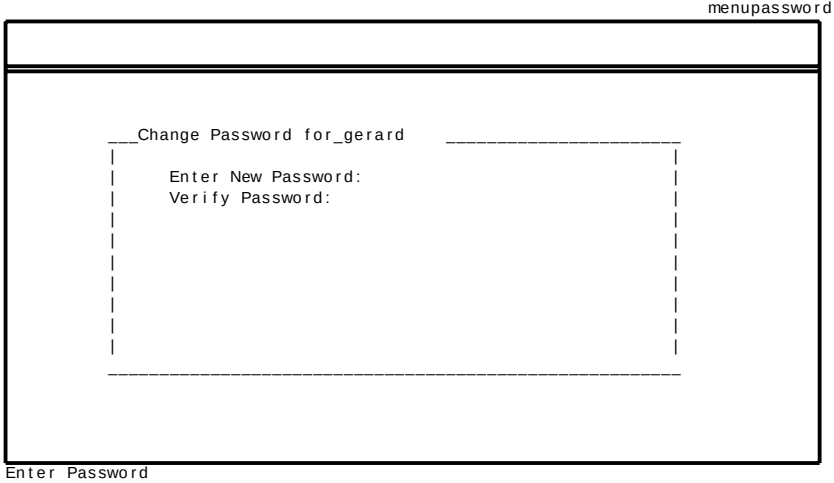
6.3 Change Password

This choice will also allow you to change user passwords but for the currently logged on user account only. Select the menu choice **Change Password** on the **Development Menu** or enter the following command.

cwmenu cwmenu2 -f chgpassword

If you would like users to change their own passwords you can place this choice on the main menu.

The screen for changing passwords is as follows.




```
Enter Password
F1   F2   F3   F4   F5   F6   F7   F8   F9   F10  F11  F12  F13  F14  F15  F16
SAVE HELP ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
```

7. THE INSTRUCTIONS SECTION

Overview

The **INSTRUCTIONS** section is where you may write C language functions to alter the default logic of a screen form, report, or menu template. The C functions will be called from userexits that you specify in other sections of your source files.

Chapter 9, TOOLFLEX, briefly describes the C function library available to your **INSTRUCTIONS** section code.

Syntax

```
INSTRUCTIONS

ufuncname ( )
{
        Ccode
        @variables
        Ccode
        $variables
        Ccode
        return( n);
}
          .
          .
          .

END
```

Description

INSTRUCTIONS is a required keyword.

ufuncname	is a C function name that is the parameter of a userexit of an earlier section of the source file.
{ }	are the required brackets that enclose the C function body.
Ccode	is C language code.
@variable	is an Infoflex structure variable embedded in the C code.
$*variable*	is an Infoflex value variable embedded in the C code.
return	is a required statement in a userexit C function. A userexit C function must return a value even if it is only 0.
n	is the required return value of the userexit C function.
END	is a required keyword.

Notes

General

- For every C function called from a userexit there must be a C function defined in the **INSTRUCTIONS** section.
- A C function called from a userexit cannot have parameters.

Field Value Variables

- Anywhere within *Ccode* you may access values from or assign values to database fields or screen form, report, or menu fields by embedding a special Infoflex variable syntax usage:

database table	$*table.field*
screen form	$*screen.fieldtag*
report subsection	$*subsect.fieldtag*
menu template	$*template.fieldtag*

  In the future we will refer to one of these variables as a **$variable**.

- You can assign a value to any **$variable** that is not associated with a character field. Here is an example using a database field:

```
$bkmaster.bkno = 10;
```

  Note that, reading this statement without the **$**, the syntax is equivalent to a C assigment to a structure element.

- Further, you can assign the value in a **$variable** to a conventional C variable. For example:

```
booknum = $bkmaster.bkno;
```

- You can also do arithmetic interchanging the two kinds of variables:

```
$customer.total =
    $customer.sale + ($customer.sale * taxrate);
```

- Money fields are stored in units of cents. Therefore, to print a money field in units of dollars, you must first divide the value by 100. When assigning a value to an Infoflex money field, be sure to the units are in cents.

- The use of **$variables** with character fields is somewhat different. Here the value is actually the address in memory of the contents of the field. You will not want to assign to this **$variable** because you will lose the address of the field data. Character field **$variables** should never change their value. What can change is the character string stored at the address. This example will change the contents of a character field:

```
strcpy( $bkmaster.agent, "John Smith");
```

  Since character **$variables** are null terminated, this works well, but the source string, "John Smith" in our example, must be the exact length of the destination field. When copying strings from and to Infoflex character fields, we recomment that you use the **TOOLFLEX** function **move** which will take care of any length mismatch problems for you.

- A substring syntax is available with character **$variables**. For example, with a database field it is:

$*table* . *field* (*start* , *length*)

  Start is the position of the first character of the substring. The first character of the field is in position 0. *Length* is the length of the substring.

- A screen form array **$variable** may be subscripted to specify the row of the associated array field. Here is an example of two array **variables**, the second is a substringed character field:

```
$bkdetail.bkno[0]
$bkdetail.bdesc[5](10,5)
```

  If the subscripting is omitted, the **$variable** applies to the current array row.

Structure Variables

- Many of the functions of **TOOLFLEX** (see Chapter 10) use one or more parameters that are database tables, screen forms, report layouts, or menu templates, or fields of any of these. These *structure* variables have a special Infoflex variable syntax. This is the syntax for structures above the level of the individual field:

database table	@*table*
screen form	@*screen*
report subsection	@*subsect*
menu	@*template*

  For an individual field, of a database table for example, the syntax is:

  @*table.field*

  In the future we will refer to one of these variables as an **@variable**.

- The length of any character field, whether it be the field of a database table, screen form, report subsection, or menu template, is available through an **@variable**. For example, with a screen form field it would be:

  @ *table.field*.fxlength

Example

```
INSTRUCTIONS

befdelete()
{

    move( @slmaster.bkno, @sldetail.bkno);
    move( NULL, @sldetail.recno);
    if ( 0 > fmfind( @sldetail, @sldetail.sldbkno, ISGTEQ) )
        return( 0);

    if ( $slmaster.bkno == $sldetail.bkno) {
        msggerr( "@Cannot delete");
        return( -1);
    }

    return( 0);
}

ae_bkdate()
{

    if ( $slinput.bkdate <  curdate ) {
        msggerr( "Date may not be prior to current date");
        return( -1);
    }

    return( 0);
}

END
```

Move, **fmfind**, and **msggerr** are functions of the TOOLFLEX function library (see Chapter 10).

8. DATABASE DATA TYPES

These are the valid data types for fields of an Infoflex database table:

CHAR

A character string can have a length of 1 to 32767 characters. When assigning a character type to a field, the length *n* is specified by this form: CHAR(*n*).

SMALLINT or SHORT

This is a whole number between -32,767 and 32,767.

INTEGER or LONG

This is a whole number between -2,147,483,647 and 2,147,483,647.

DECIMAL

This is the machine independent representation of the decimal number of up to 32 digits of precision. In the form DECIMAL(*m*,*n*) *m* is the total number of digits in the number, *n* is the number of digits right of the decimal point. DECIMAL without parameters defaults to DECIMAL(16).

SMALLFLOAT

This is the data type corresponding to the float C data type on your machine.

FLOAT or DOUBLE

This is the data type corresponding to the double C data type on your machine.

MONEY

The MONEY data type has the structure as the DECIMAL type except that a money field in a screen or report will by default display with a dollar sign. The MONEY type is parameterized the same way as the DECIMAL, except MONEY(*m*) is equivalent to DECIMAL(*m*,2), and MONEY without parameters is equivalent to DECIMAL(16,2).

SERIAL

Each newly inserted record with a value of 0 for the SERIAL field will receive a value 1 greater than the SERIAL field of the previous record inserted. By default the SERIAL field of the first record inserted is 1. If the SERIAL field was assigned with SERIAL(*n*), then the first record will have the value *n*. There can be at most one SERIAL field per table record. The maximum serial is 2,147,483,647.

DATE

In a table a DATE is stored as the number of days since December 31, 1899. It has equivalent size to the INTEGER type. By default, dates are displayed to screen and report fields in the form of *mm*/*dd*/*yy*. They can be input to screen fields in the same format.

TIME or MTIME

In a table TIME or MTIME is stored as the number of elapsed seconds. It has equivalent size to the LONG type. By default, time fields are displayed to screen and report fields in the form of *HH:MM:SS*. By default, mtime fields are displayed to screen and report fields in the military (24 hour clock) form of *HH:MM:SS*. They can be input to screen fields in the same format. TIME or MTIME field types may not be used within the SQLflex cammands but may be used within Screen or Report forms.

9. ENVIRONMENT VARIABLES

An environment variable is a variable that maintains its definition between programs or operating system commands. This chapter describes each of the environment variables that are used by the Infoflex system. The Application Set-Up topic in Chapter 2 describes the **fxsetenv** utility used for automatically setting up these variables for Infoflex development.

FXDIR

FXDIR is the full path name for the base directory of the Infoflex development system.

FXBIN

FXBIN is the full path name of the directory containing the binary **.pic** files that define the screen and report formats of an Infoflex application. The special file **sysmsg.flx** that contains all the standard messages of any Infoflex application must also reside in this directory. The **fxmkapp** utility, described in The Application Set-Up topic in Chapter 2, will install **sysmsg.flx**.

FXDATA

FXDATA is the full path name of the Infoflex database directory.

FXHELP

FXHELP is the full path name of the directory containing the application help files.

FXEDIT

FXEDIT defines the name of the editor program used in editing Infoflex source files from the Infoflex Development Menu. It will also be the editor used by the Infoflex **prntflex** utility.

Initially **FXEDIT** is undefined, in which case the **vi** editor will be used on UNIX/XENIX systems and the **e** editor on DOS systems.

FXPRINT

FXPRINT defines the printer control file to be used when printing. Refer to the Printer Setup chapter for further explanation.

FXPRT

FXPRT controls printer configuration. Refer to the Printer Setup chapter for further explanation.

FXDATE

FXDATE defines the date format used throughout the screens and reports of an Infoflex application. In the format definition **MM** represents the month, **DD** represents the day of the month, and **YY** represents the year. **MM/DD/YY** is an example of a definition for **FXDATE**.

Initially **FXDATE** is not defined, and the default format of **MM/DD/YY** is used.

10. TOOLFLEX

10.1 Overview

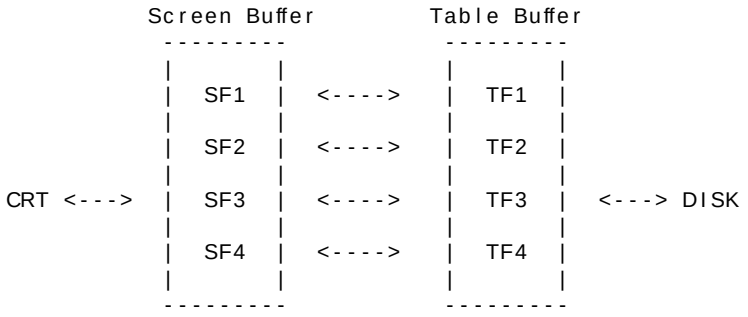
This chapter describes all of the special C language functions available to your **INSTRUCTIONS** sections. These functions provide the following general capabilities:

- Table management for adding, changing, or deleting database records.
- Screen management for prompting and displaying on a variety of terminal types.
- Data flow management between and among screen and table data buffers.
- Branching to external programs.

The first three topics within this chapter, Buffer Usage, Global Variables, and Function Arguments, provide a good overview before dealing with the specific functions of the TOOLFLEX library.

10.2 Data Buffers

To better understand TOOLFLEX functions, a brief explanation about Infoflex's data buffers is essential. There are two data buffers used when moving data between the video display and the database. These are the screen buffer and the table buffer and are pictorially shown below:



SF*n* represents a screen field. **TF***n* represents a database table field.

Screen Buffer

The screen buffer holds data for all fields belonging to a single screen image. Fields in this buffer are stored one after another in a continous string and are in C internal format. Also, character fields are null terminated. Data is displayed from or saved to this buffer whenever you use a screen management functions described later in this chapter.

Table Buffer

The table buffer holds data for all fields belonging to a single database table. Fields in this buffer are stored one after another in a continous string and are in C-ISAM format (see C-ISAM manual section IV). Also, character fields are null terminated. Data from a database table is written from and read to the table buffer whenever you use the table management functions descibed later in this chapter.

How data entered on the screen is saved in the database and vice versa involves a series of steps:

1) The screen prompts for the data using a screen management function (see the topic Screen/Report Management Functions).
2) Data is moved from the screen buffer to the table buffer using a data flow management function (see the topic Data Flow Management Functions).
3) Data is written to a database table using a table management function. (see the topic Table Management Functions).

On the other hand, to display data on the screen that is stored in a database table, the reverse flow takes place. In this case the program performs the following steps:

1) Data is read from a database table using a table management function.
2) Data is moved from the table buffer to the screen buffer using a data flow management function.
3) Data is displayed using a screen management function.

10.3 Global Variables

This topic describes the global variables that you may use in your **INSTRUCTIONS** section C code. Most of these variables are informational and must not be changed by the user. The variables that are user definable will be indicated as such.

General

int flexmode
: has the value of the current mode, which is one of these defines:
ADDMODE
CHANGEMODE
SEARCHMODE
HELPMODE

int flexkey
: has the value of the last action key pressed (see the the include file **fxcrt.h** for possible key values).

long curdate
: is to the current date. It is the number of days since December 31, 1899.

char msgbuf[200]
: is a buffer available for user definable messages. You may assign values to this buffer.

int *global.gargc
: is the number of arguments in the program command line.

char *global.gargv[]
: points to the program command line argument list.

Database Pointers

DBHEAD *global.gdbhead
: points to the table associated with the current screen portion, header or array, as defined in the SELECT section.

DBINDEX *global.gdbindex
: points to index of the table associated with the current screen portion, header or array, as defined in the SELECT section.

DBFIELD *global.gdbfield
: points to the table field associated with the current screen field. Accessable from field level userexits (i.e. beforedit, afteredit, etc.).

Screen Pointers

SCRHEAD *global.gscrhead
: points to the current screen portion, header or array.

SCRFIELD *global.gscrfield
: points to the current screen field. Accessable from field level userexits (i.e. beforedit, afteredit, etc.).

10.4 Function Arguments

In following topics, some conventions are used with function argument names. The following is a brief description of these conventions:

pscrhead	is a pointer into an array of screen header structures of type **SCRHEAD**. The **SCRHEAD** structure contains information about a screen.
pscrfield	is a pointer into an array of screen field structures of type **SCRFIELD**. The **SCRFIELD** structure contains information about a screen field.
pdbhead	is a pointer into an array of table header structures of type **DBHEAD**. The **DBHEAD** structure conatins information about a table.
pdbindex	is a pointer into an array of table index structures of type **DBINDEX**. The **DBINDEX** structure contains information about a table index.
pdbfield	is a pointer into an array of table field structures of type **DBFIELD**. The **DBFIELD** structure contains information about a table field.
pfield	is a pointer into an array of screen or table field structures of type **FIELD**. The **FIELD** structure contains information about a screen or table field.
phead	is a pointer into an array of screen or table header structures of type **HEAD**. The **HEAD** structure contains information about a screen or table.

For functions requiring any of the above arguments, you should use the **@variables** as described in the Chapter 7, THE INSTRUCTIONS SECTION. For run-time efficiency the **@variables** are converted to structure pointers by the Infoflex pre-processor.

10.5 Table Management Functions

This topic describes the file management functions. These functions will enable you to read and write fixed length records from and to a database table.

Below is a list of file management functions and a brief description of each. Note the correspondence between the C-ISAM set of functions and the ones listed below. For a more detailed description of these functions, refer to the TOOLFLEX FUNCTION REFERENCE MANUAL.

In general, these functions will not effect the system catalogs (database dictionary). To alter the system catalogs, you should use **SQLFLEX**.

fmaddindex(pdbhead, pdbindex)
: adds an index to a table.

fmbldall(pdbhead)
: builds a table and all its indices.

fmbuild(pdbhead, pdbindex, mode)
: creates a table.

fmclose(pdbhead)
: closes a table.

fmcurchk(pdbhead)
: tests if the read cursor of a table is positioned on a record.

fmcurclr(pdbhead)
: Clears the read cursor of a table.

fmdelcurr(pdbhead)
: deletes the current record.

fmdelete(pdbhead)
: deletes a record. **Fmdelete** will only work with tables created with the **fmbuild** function. It will not work with an SQLFLEX created table.

fmdelindex(pdbhead, pdbindex)
: removes an index.

fmdelrec(pdbhead, recno)
: deletes a record from the table identified by its physical record number.

fmdictinfo(pdbhead, pdictinfo)
: gets table parameters.

fmerase(pdbhead)
: removes a table.

fmerrmsg(pdbhead)
: displays an error message based on the last file management call.

fmfind(pdbhead, pdbindex, mode)
: finds a record based on mode and index. Does a combination of **fmstart** and **fmread**.

fmflush(pdbhead)
: flushes data and indexes to disk.

fmindexinfo(pdbhead, pdbindex, pkeydesc)
: gets information about a table's indices.

fmload(tablelist)
: dynamically loads table information into memory.

fmlock(pdbhead)
: Creates a lock on the entire file.

fmopen(pdbhead, mode)
: opens a table.

fmread(pdbhead, mode)
: reads a record into the table buffer.

fmrelease(pdbhead)
: unlocks a table record.

fmrename(oldname, newname)
: renames a table.

fmrewcurr(pdbhead)
: rewrites the current record from the table buffer.

fmrewrec(pdbhead, recno)
: rewrites a record from the table identified by its physical record number.

fmrewrite(pdbhead)
: rewrites a record from the table buffer. **Fmrewrite** will only work with tables created with the **fmbuild** function. It will not work with an SQLFLEX created table.

fmsave(pdbhead)
: saves the record defined in the table buffer. **Fmwrite** is called if the last **fmfind** call was unsuccesful. **Fmrewcurr** is called if the last **fmfind** call was successful.

fmsetserial(pdbhead)
: sets the serial number in the table buffer.

fmsetunique(pdbhead, recnum)
: sets the value of the internal unique identifier.

fmstart(pdbhead, pdbindex, keylength, mode)
: positions the read cursor in a table.

fmstructview(pdbhead, pdbview, vwnum, vwstruct)
: maps a set of database fields to a structure for future table management calls.

fmuniqueid(pdbhead, recnum)
: gets the next unique number for a table.

fmunlock(pdbhead)
: Unlocks a file previously locked with **fmlock()**.

fmwrcurr(pdbhead)
: writes a table record and move the table cursor.

fmwrite(pdbhead)
: adds a record to a table. **SERIAL** type fields are automatically incremented.

fxasave( )
: is the default function for saving an array screen line to a table.

fxssave( )
: is the default function for saving a non-array screen data to a table.

lookup(pdbhead, pdbindex)
: looks up in a table based on the specified index.

10.6 Screen/Report Management Functions

This topic describes the screen management functions. These functions will enable you to:

1) Initialize your Infoflex program to handle different terminal types.
2) Prompt for data from the screen.
3) Display data on the screen.
4) Display literals and messages to the screen.
5) Set-up function key validation

Below is a list of the screen management functions subcategorized by the above capabilities with a brief description of each function. For a more detailed description of these functions, refer to the TOOLFLEX FUNCTION REFERENCE MANUAL.

Program Initialization and Termination

flexload(picname)
: initializes the terminal and loads a **.pic** file.

flex(argc, argv)
: same as flexload() plus runs the flex program.

flexcmd(command)
: same as flex().

fxabort(status)
: exits a Infoflex program gracefully.

fxinit( )
: initializes an Infoflex program for a particular terminal type.

Data Prompting

fxaccept(pscrfield)
: prompts for a screen field defined in the **SCREEN** section.

getkey( )
: prompts for a single key press.

inyesno(row, col, msg)
: prompts for a **Y** or **N** response.

prompt(row, col, length, dec, type, attr, fmt, buffer)
: prompts for data from any position on the screen (not into a field variable).

Data Display

getxypos(pscrfield, prow, pcol)
: gets the CRT row and starting column position of a screen field.

scrollpage(mode)
: scrolls the array portion of the screen.

tclrall(pscrhead)
: clears all screen fields for the screen header or array.

tclrfld(pscrfield)
: clears a screen field.

tclrrec(pscrhead)
: clears all screen fields for the screen header or the current row of the array.

tclrrng(pscrfield, pscrfield)
clears a range of screen fields.

tmapfld(pscrfield)
displays a screen field from the screen buffer.

tmaprec(pscrhead)
displays all screen fields from the screen buffer.

tmaprng(pscrfield, pscrfield)
displays a range of screen fields from the screen buffer.

Literal and Message Display

bflush( )
Flush internal I/O buffer to the terminal screen.

boxline(urow, lcol, brow, rcol)
displays a box using a single line border. **Bflush** must be called for this function to take effect.

boxrev(urow, lcol, brow, rcol)
displays a box with a reverse attribute background. **Bflush** must be called for this function to take effect.

bshow(msg, attr)
buffers a message to display on the screen.

bshowxy(row, col, msg, attr)
buffers a message to display at any position on the screen.

clrbox(urow, lcol, brow, rcol)
clears a box. **Bflush** must be called for this function to take effect.

clreol( )
clears to the end of the current screen line. **Bflush** must be called for this function to take effect.

clreos(row, col)
clears to the end of the screen. **Bflush** must be called for this function to take effect.

clrpage(pscrhead)
clears a screen form region. **Bflush** must be called for this function to take effect.

clrrng(row, nlines)
clears a range of screen rows.

clrscr( )
clears the entire screen. **Bflush** must be called for this function to take effect.

gotoxy(row, col)
positions the screen cursor to the specified row and column. **Bflush** must be called for this function to take effect.

graphout(row, col, length, graphmacro)
draws a horizontal or vertical line.

message(row, col, msg, attr)
clears a line and displays a message on the screen.

msgcomment(msg)
displays a message to the comment line.

msgerr(msg)
displays a message to the error line.

msggerr(msg)
displays a message to the error line and waits for a key press.

msgstat(msg)
displays a message to the status line.

msgwait(msg)
: displays a blinking message to the status line.

page(pscrhead)
: displays all screen literals.

putkey(row, col, key)
: displays a single character on the screen. **Bflush** must be called for this function to take effect.

repaint( )
: repaints the screen literals and fields.

setcursor(mode)
: Turn the screen cursor on or off.

show(msg, attr)
: displays a message on the screen.

showxy(row, col, msg, attr)
: displays a message at any position on the screen.

skipto(pscrfield)
: specifies the next screen field to take input.

Function Key Validation

keychglabel(key, label)
: changes a function key label.

msgfunc(msg)
: displays a message to the function key label line and makes that the new function key label.

msgnfunc( )
: displays the function key number label: **F1 F2 F3** ...

Reports Only

rptline(buffer)
: outputs a linefeed character to a report.

rptprint()
: outputs a record according to the REPORT section.

rptgetline( )
: returns current report line number.

rptlneed(n)
: requests a number of report output lines for the page.

rptformfeed( )
: outputs the new page character to a report.

Miscellaneous

chkent(pscrfield)
: check if a field is data enterable.

modoffrng(pscrfirst, pscrlast)
: turns off the modify flag for a range of screen fields.

modonrng(pscrfirst, pscrlast)
: turns on the modify flag for a range of screen fields.

nodisplay(pscrfield)
: sets the nodisplay flag for a screen field.

nolookup(pscrfield, type)
: sets the nolookup flag for a screen field.

sfswap(pscrfield, pscrfield)
: swap two screen fields.

skip(pscrfield)
: marks a field so it is skipped during data entry.

unnodisplay(pscrfield)
: turns the nodisplay flag off for a screen field.

unskip(pscrfield)
: marks a field so it is not skipped during data entry.

10.7 Data Flow Management Functions

This topic describes the data flow management functions. These functions will enable you to move data between and among the screen and table buffers as well as move data between the buffers and C variables. The Data Buffers topic in this chapter describes these buffers in detail.

Below is a list of data flow management functions and a description of each. For a more detailed description of these functions, refer to the TOOLFLEX FUNCTION REFERENCE MANUAL.

Screen Buffer

buftostr(s1, s2, type, length, dec, fmt)
: converts a value to a string.

getsf(pvalue, pscrfield)
: gets the field value from the screen buffer.

SCRFIELD *getsfp(pscrhead, name)
: gets a screen field pointer.

SCRHEAD *getshp(name)
: gets the pointer to a screen header.

putsf(pvalue, pscrfield)
: puts a value into a field of the screen buffer.

sclrfld(pscrfield)
: clears a field in the screen buffer.

sclrrec(pscrhead)
: clears all fields in the screen buffer.

sclrrng(pscrfirst, pscrlast)
: clears a range of fields in the screen buffer.

strtobuf(s1, s2, type, length, fmt)
: converts a string to a value.

Table Buffer

dclrfld(pdbfield)
: clears a field in the table buffer.

dclrrec(pdbhead)
: clears all fields in the table buffer.

dclrrng(pdbfield, pdbfield)
: clears a range of fields in the table buffer.

getdf(pvalue, pdbfield)
: gets a field value from the table buffer.

DBFIELD *getdfp(pdbhead, name)
: gets a table field pointer.

DBHEAD *getdhp(name)
: gets the pointer to a table header.

putdf(pvalue, pdbfield)
: puts a field value into the table buffer.

General Buffer/Variable

datestr(datenum, datestr, format)
: converts a number of days since December 31, 1899 to date string format. The format is specified by a format string.

dmapfld(pdbhead, pscrfield)
: copies a field from the screen buffer to its related table buffer field.

dmaprec(pdbhead, pscrhead)
: copies all fields from the screen buffer to their related table buffer fields.

dmaprng(pdbhead, pscrfield, pscrfield)
: copies a range of fields from the screen buffer to their related table buffer fields.

double fxround(value, places)
: rounds a value to the specified number of places.

gettime(numtime)
: gets the current time in number of seconds since midnight.

gettoday(numdate)
: gets today's date in days since December 31, 1899.

isempty(pfield)
: tests for a 0 or null value in a field.

ismodfld(pfield)
: tests if a field has been modified.

ismodrng(pfield, pfield)
: tests if a range of fields has been modified.

isnull(pfield)
: returns 1 if the field value is null.

iszero(pfield)
: returns 1 if the field value is zero.

move(pfield, pfield)
: copies the data of one field to another. This function will convert data types if necessary. In addition, one of the arguments may be a pointer to a C variable provided it is of the same type and length.

rdatestr(datenum, datestr, type)
: converts a number of days since December 31, 1899 to date string format. The format is specified by a macro.

rstrdate(datestr, datenum, type)
: converts a date format string to the number of days since December 31, 1899. The format is specified by a macro.

rstrtime(s, secs, type)
: converts a string to an internal time format.

rtimestr(secs, s, type)
: converts an internal time format to a string.

setnull(pfield)
: sets the value of field to null.

setzero(pfield)
: sets the value of field to 0.

smapfld(pdbhead, pscrfield)
: copies a field from the table buffer to its related screen buffer field.

smaprec(pdbhead, pscrhead)
: copies all fields from the table buffer to their related screen buffer fields.

smaprng(pdbhead, pscrfield, pscrfield)
: copies a range of fields from the table buffer to their related screen buffer fields.

strcenter(s, totlen)
: centers a string within a given length.

char *strcompress(s)
: deletes leading and trailing white space from a string.

strdate(datestr, datenum, format)
: converts a date format string to the number of days since December 31, 1899. The format is specified by a format string.

char *strfind(s, t, comp)
: finds a substring within a string.

char *strltrim(s, c)
: trims a character from the left of a string.

char *strscan(s)
: skips leading white space.

char *strtrim(s, c)
: deletes a trailing character from a string.

sysdate(buffer)
: gets the system date.

systime(buffer)
: gets the system time.

10.8 Program Branching

fxcallv(argc, argv)
: calls another program using main-like parameters. The calling program is returned to.

fxchain(argc, argv)
: chains to another program. The calling program is not returned to.

fxsystem(command)
: does a system call.

10.9 Main Function

When creating an Infoflex program, a default **main** function is automatically provided to you. If you choose to write your own **main** function, you will need to do the following:

- The **flexload** function (under the Screen Management Functions) must be called prior to using any other TOOLFLEX function. **Flexload** sets up the terminal and loads **.pic** file information.
- The **fxabort** function (under the Screen Management Functions) must be called at the end of your program to restore terminal settings and close any open files.

Example

```
INSTRUCTIONS

main( argc, argv)
int argc;
char *argv[];
{
    flexload( argv[1]);   /* argv[1] is the .pic file */

    Rest of program

    fxabort( 0);
}

END
```

10.10 Error Codes

When an error condition arises in the execution of a standard C function or a TOOLFLEX function, the global variables **errno** and/or **iserrno**, may be set to an error code.

Iserrno will be set whenever an error occurs with the file management routines (functions which start with **fm**). A listing of the possible values for **iserrno** and there meanings can be found in chapter E-Runtime Errors.

11. THE HELP SYSTEM

11.1 Overview

The Infoflex on-line help facility provides information at many levels, from the most general help down to field specific help.

The help files are simple text files that are created in an editor of your choice. The **FXHELP** environment variable defines the directory in which the help files of your application reside.

The following topics show how to properly name and format these files so they can be appropriately accessed at runtime.

11.2 Levels of Help

The Help facility displays twenty lines of help file text at a time. The other 4 available lines are used for titles and function key labels. To control the page breaks of the help text, you will use the special character **%** placed at the begining of a line.

The following are the different levels of help and how they are set up.

Application Wide

Help information about the application as a whole is placed in the file **helpgen.hlp**.

Module Wide

Help information that applies to a group of source files (or module) is placed in a file called *xx***gen.hlp**, where *xx* are the first two characters of the source filenames. To use this help level effectively, source filenames belonging to the same module should begin with the same two characters.

Source

Source help is the help information that applies to a single source file. Help information about all screens and their fields are placed in this file. The name of the help file will be the same name as the source file with the extension **.hlp**.

There are a couple of simple rules governing the organization of the source level help file. This organization will allow you to provide help text for an unlimited number of screens and fields within any source file.

The screen level help within your help file is designated by placing the special character **!** in the first position of a line followed by the screen name. The help text for this screen should follow on the next line and may be any number of lines.

The field level help will come after the screen level help. To designate where the help text begins for a field, you will place an **@** in the first position of a line followed by the screen's *tagname*. The field level help text should start on the next line.

Below is a simple example of a source help file:

```
!custscreen
     This screen is for entering customer
     information.  This screen will allow
     you to update, delete, or add. Customers
     providing you have the proper
     authorization.
%
@custcode
     This is the Customer code.
     This code is required and
     must be numeric and no more
     than 10 characters in length.

@custname
     This is the Customer name.
     The name is alphanumeric and
     should always be entered in uppercase.

@custphone
     The phone number should include
     the area code.
```

Action Keys

This help information is as general as the application wide help and can be accessed from any screen form of the application. This help provides instruction as to the general function of all the special keys of the application, including cursor move keys and function keys. A canned **helpfunc.hlp** file is supplied with every distribution of the Infoflex Development System. You may modify this file in the help directory of your particular application.

User Notes

User notes help information is any screen form specific notes that anyone may want to add to the file *source***.not**, where *source* is the name of the source file.

11.3 Help At Run-Time

Here are the run-time features of the Infoflex help system:

- The help subsystem is accessed by pressing the **HELP** key while in a screen form.
- First any help information is displayed for the specific form field that was cursor addressed at the time that the **HELP** key was pressed.
- With a help screen displayed, the function key ruler at the bottom of the video display is:

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
EXIT	----	JUMP	----	----	----	PREV	NEXT	----	----	----	----	----	----	----	--

- Pressing the **NEXT** key pages through the help available screens. From the field specific help it will move on to the action key help, then to the user notes, then to the application wide, module wide, and finally screen help before it loops back to the field specific help again. The **PREV** key reverses the direction of the help paging.
- The **JUMP** key displays a prompt that allows the user to select the type of help he wants.
- At the time that the **HELP** key was pressed if the cursor addressed a screen form field that has a **tablehelp** attribute, then a table help screen is additionally available through the **JUMP** prompt. See the next chapter for a full description of the **tablehelp** attribute.
- **EXIT** or **ESC** (**ESC ESC** on UNIX or XENIX systems) returns to where the user left off in the calling screen form.

12. SQLFLEX

SQLFLEX is the Infoflex language and procedure for creating and modifying the structure of your Infoflex databases and querying their contents.

To execute an SQLFLEX operation you must create a text file *script***.sql** which contains statements in the language of SQLFLEX. You then invoke *script***.sql** from your operating system prompt with:

fxsql *script*

The database effected is the one defined by the **FXDATA** environment variable.

Every SQLFLEX statement in *script***.sql** must be terminated with a semicolon, **;**.

Below is a list of all the SQLFLEX statements and brief description of each. Each statement will be described in detail in the subsequent topics of this chapter.

alter table	changes the file structure of a table.
create database	creates a database.
create index	creates an index for a table.
create table	creates a table.
delete	delete records from a table.
drop database	removes a database.
drop index	removes an index from a table.
drop table	removes a table from a database.
info	provides info about the structure of a database.
insert	insert a record into a table.
load	load data into a table from a text file.
rename column	changes the name of a table field.
rename table	changes the name of a database table.
select	queries data from one or more tables.
unload	unload data from a table into a text file.
update	update data of records in a table.

12.1 ALTER TABLE

Overview

The **alter table** statement modifies the field structure of a table. You may add fields, remove fields, or change their data type.

Syntax

```
alter table table
    {   add( newfield newdtype [ before oldfield ], . . .)
      | drop( oldfield, . . .)
      | modify( oldfield newdtype [ not null ], . . .)
    } . . .
```

Description

alter table	are required keywords.
table	is the required name of a table of the current database.
add	is a keyword used to add a new field to *table*.
newfield	is the name of the field to be added.
newdtype	is *newfield*'s data type specification or, for **modify**, *oldfield*'s new data type specification.
before	is an optional keyword that specifies the existing field of *table* before which *newfield* will be inserted. Without a **before** clause, *newfield*s are added to the end of the sequence of fields.
oldfield	is the name of an existing field of *table*.
drop	is a keyword used to remove an *oldfield*.
modify	is a keyword used to change the data type or length of an *oldfield*.
not null	are optional keywords specifying that a field may never have a null value.

Notes

- You may use any number of **add**, **drop**, or **modify** clauses and in any order. Separate the clauses with commas. The actions are performed in the specified order, and if any fail, the entire operation is canceled.
- You may not add a **serial** field to a table with records.
- You cannot modify a field to **not null** if existing records have null values in that field.
- Character to numeric data type conversions will lose non-numeric information in the original character fields.
- For *newdtype* see Chapter 8, DATABASE DATA TYPES.

Example

```
alter table bkmaster
    add( agent char(4) before status),
    drop(  comamount, refamount),
    modify( postdate date not null);
```

12.2 CREATE DATABASE

Overview

The **create database** statement creates a database.

Syntax

```
create database dbname
```

Description

create database are required keywords.

dbname is a required name of the database to be created.

Notes

- This statement will create the database directory *dbname***.dbs** in the current directory. All the initial system catalog tables will be created.

Example

```
create database trvagen
```

12.3 CREATE INDEX

Overview

The **create index** statement creates an index for a table.

Syntax

```
create [ unique ] index indexname
    on table( field [asc | desc] [ , . . . ] )
```

Description

create index	are required keywords.
unique	is an optional keyword which prevents more than one record from having the same value for *indexname*.
indexname	is a required name of the index. There must be a unique identifier for every index of a database.
on	is a required keyword.
table	is the name of the table for which the index is being created.
field	is the name of a field of *table* that is used in the index. The index may be made up of one or more fields of *table*. If more than one, the *field*s are comma separated.
asc	is an optional keyword to specify an ascending ordered field. This is the default condition.
desc	is an optional keyword to specify a descending ordered field.

Example

```
create index bkmagent
    on bkmaster( agent, bookdate desc, bkno);
```

12.4 CREATE TABLE

Overview

The **create table** statement creates a table in the current database.

Syntax

```
create table tablename
    (field dtype [ not null ], . . .)
```

Description

create table	are required keywords.
tablename	is the name of the database table being created.
field	is the name of a field of *table*.
dtype	is the data type of *field*.
not null	are optional keywords specifying that a field may never have a null value.

Notes

- Table names must be unique within a database.
- Field names must be unique within a table, but the same field name can be used in different tables.
- If there is more than one field in a table, the field specifications are separated by commas.
- A table can have no more than one **serial** type field.
- For *dtype* see Chapter 8, DATABASE DATA TYPES.

Example

```
create table bkmaster (
    bkno          serial(50),
    name          char(20),
    bookdate      date,
    passno        integer not null,
    reference     char(6),
    agent         char(4),
    status        char(1),
    postdate      date not null,
    salamount     money,
    comamount     money,
    refamount     money
);
```

12.5 DELETE

Overview

The **delete** statement deletes one or more records from a database table.

Syntax

```
delete from table
    [ where wcondition ]
```

Description

delete from	are required keywords.
table	is the name of the table from which to delete records.
where	is an optional keyword for a clause that specifies selection criteria that may reduce the scope of the records deleted.
wcondition	is the boolean expression of the **where** clause.

Notes

- A **delete** statement without a **where** clause will delete all records of *table*. In this case the user is asked if he is sure he wants to delete all records of the table.
- The syntax for *wcondition* is exactly the same as for the **select** statement.
- The **delete** statement locks *table* from other users' modifications during the delete operation.

Example

```
delete from alltypes
     where fint = 1000;
```

12.6 DROP DATABASE

Overview

The **drop database** statement removes a database.

Syntax

```
drop database dbname
```

Description

drop database	are required keywords.
dbname	is the name of the database to be removed.

Notes

- WARNING: all data within the database will be deleted with this statement.
- This statement removes the directory *dbname***.dbs** from the current directory. On UNIX and XENIX systems you must own the directory to remove it.

Example

```
drop database bkmagent;
```

12.7 DROP INDEX

Overview

The **drop index** statement removes an index from the database.

Syntax

```
drop index indexname
```

Description

drop index	are required keywords.
indexname	is the name of index to be removed.

Example

```
drop index bkmagent;
```

12.8 DROP TABLE

Overview

The **drop table** statement removes a table and its associated indices from a database.

Syntax

```
drop table tablename [ onerror [ warning ] ]
```

Description

drop table	are required keywords.
tablename	is the name of the table to be removed.
onerror	is an optional keyword where if *tablename* does not exist, **drop table** will not produce a fatal error condition for the SQLFLEX script.
warning	is an optional keyword that may be used with the **onerror** clause. A non-fatal warning message is displayed if *tablename* does not exist.

Notes

- WARNING: all data within *tablename* will be deleted with this statement.

Example

```
drop table bkmaster;
```

12.9 INFO

Overview

The **info** statement provides information about the database structure.

Syntax

```
info {  tables
      | columns for tablename
      | indices for tablename
     }
```

Description

info	is a required keyword.
tables	is a keyword requesting a list of the tables of the database.
columns for	are keywords requesting a list of the column of field names for a table.
indices for	are keywords requesting a list of the indices for a table.
tablename	is the name of the table for which field or index information is being requested.

Notes

- Information on a field will also provide the field type and whether a null value is permitted for the field.
- Information on an index will also provide the table fields comprising the index and whether the index allows duplicate values.

Examples

```
info tables

info columns for bkmaster;

info indices for bkdetail;
```

12.10 INSERT

Overview

The **insert** statement creates database table records.

Syntax

```
insert into table [ ( fieldlist ) ]
      {  values constantlist
       | selectstmt
      }
```

Description

insert into	are required keywords.
table	is the name of the table in which to add records.
fieldlist	an optional comma separated list of field names. Each field must be a field of *table*.
values	is a keyword.
constantlist	a comma separated list of constant values that will be inserted into the corresponding fields of *fieldlist.*
selectstmt	a **select** statement used in place of the **values** clause to create records by selecting field values from other fields of the database.

Notes

- If *fieldlist* is omitted, then all fields of the created record will have values inserted and in the order that the SQLFLEX **info** statement describes.
- The values of the **values** clause or those selected using the **select** statement must have compatible data types with the corresponding fields being inserted.
- Decimal precision loss or string truncation may occur and is reported as warning messages when it does. The warning messaging can be turned off with the **-nw** command line option to **fxsql**.
- Numeric constants of *constantlist* can be represented as strings, that is, bracketed with quotation marks.
- A **select** statement used in an **insert** cannot have an **into temp** or **order by** clause.
- Serial type fields of inserted records will receive the next serial value for the table if the serial field is omitted from *fieldlist* or, when it is included, the corresponding value in *constantlist* is 0.

Examples

```
insert into alltypes (
        fchar2,       fsmallint,   fshort,      fint,
        flong,        fdec2,       fdec3,       fsmallflt,
        ffloat,       fdouble,     fmoney,
        fdate,        ftime,       fmtime
    )
    values (
        "A",          2,           3,           4,
        50000,        12.34,       123.456,     123.4,
        123456.789,   987654.321,  999.99,
        "01/01/89",   "08:17:45",  "14:07:55"
    );
```

12.11 LOAD

Overview

The **load** statement loads data into a database table from a specially formatted ASCII text file. This is useful in transfering information between computers or databases in a machine independent format.

Syntax

```
load from "loadfile"
     insert into table [ ( fieldlist ) ]
```

Description

load from	are required keywords.
loadfile	is the required absolute or relative path name of a text file containing the data to be loaded.
insert into	are required keywords.
table	is the name of the table in which to load records.
fieldlist	an optional comma separated list of field names. Each field must be a field of *table*.

Notes

- Each line of *loadfile* represents the data of a single record. The data of each field is terminated with the vertical bar character, |. Fields do not have to be fixed width, and character strings are not quoted.
- If *fieldlist* is omitted, then all fields of the created record will expect values from *loadfile* and in the order that the SQLFLEX **info** statement describes.
- The values of in *loadfile* must have compatible data types with the corresponding fields being loaded.
- Decimal precision loss or string truncation may occur and is reported as warning messages when it does. The warning messaging can be turned off with the **-nw** command line option to **fxsql**.
- Serial type fields of loaded records will receive the next serial value for the table if the serial field is omitted from *fieldlist* and therefore *loadfile*, or, when it is included, the corresponding field in *loadfile* is 0.

Example

```
load from "alltypes.ld"
    insert into alltypes (
            fchar2,      fsmallint,   fshort,
            flong,       fdec2,       ffloat,
            fmoney,      fdate,       ftime
        );
```

will successfully load the following file **alltypes.ld**:

```
A|2|3|50000|12.34|123456.789|999.99|01/01/89|08:17p|
```

12.12 RENAME COLUMN

Overview

The **rename column** statement changes the name of a database table field. Column is the standard SQL term for a table field.

Syntax

```
rename column table.oldfield to newfield
```

Description

rename column	are required keywords.
table	is the required name of the table of the field to be renamed.
oldfield	is the old name of the field to be renamed.
to	is a required keyword.
newfield	is the new name of the field to be renamed.

Example

```
rename column bkmaster.booknum to bkno;
```

12.13 RENAME TABLE

Overview

The **rename table** statement changes the name of a database table.

Syntax

```
rename table oldtable to newtable
```

Description

rename table	are required keywords.
oldtable	is the old name of the table to be renamed.
to	is a required keyword.
newtable	is the new name of the table to be renamed.

Example

```
rename table bookmast to bkmaster;
```

12.14 SELECT

Overview

The **select** statement queries information from one or more tables of a database.

Syntax

```
select [ all | unique | distinct ] selectlist from tablelist
    [ where wcondition ]
    [ group by grouplist ]
    [ having hcondition ]
    [ order by sortfield [ desc ] [ , . . . ] ]
    [ into temp temptable ]
```

Description

select	is a required keyword.
all	is a optional keyword that specifies the default selection result, all rows satisfying the **where** clause, including any duplicate rows.
unique	is a optional keyword that will eliminate duplicate rows in the query results.
distinct	is a optional keyword and is a synonym for **unique**.
selectlist	is a list of constant expressions and/or field expressions separated by commas. In the future we will refer to a SQLFLEX field expression as a **fieldexp**.
from	is a required keyword.
tablelist	is a list of one or more table names separated by commas.
where	is an optional keyword for a clause that specifies selection criteria that may reduce the scope of the query results.
wcondition	is the boolean expression of the **where** clause.
group by	are optional keywords to produce a single row of results for selected records with the same values for *grouplist*.
grouplist	is a list of field names separated by commas.
having	is an optional keyword that applies one or more qualifying conditions to groups.
hcondition	is the boolean expression of the **having** clause.
order by	are optional keywords that will allow you to sort your query results.
sortfield	is a field of *selectlist* that will be sorted for the **order by** clause. Where there is more than one *sortfield*, they will be separated by commas.
desc	is an optional keyword to specify that the sort order of a *sortfield* be reversed.
into temp	are optional keywords to specify that the results of the **select** should be output to a temporary table. By default a **select** outputs to the screen.
temptable	is the name of the temporary table of the **into temp** clause.

Notes

General

- There are two general classifications of **fieldexp**. First, a **fieldexp** can be a field or a calculation involving one or more fields of a single record. This is called a single record **fieldexp**. The second kind of **fieldexp** involves

operations over a number of records of a table. This is called an aggregate expression and must use one or more of the aggregate functions. These functions will be discussed in detail under a later topic, Aggregate Functions.

- The boolean expression of *wcondition* or *hcondition* can involve comparisons with constant expressions as well as **fieldexp**s. A **fieldexp** in *wcondition* must be a single record **fieldexp**. A **fieldexp** in *hcondition* must be an aggregate expression. Boolean expressions will be taken up in the next topic, Boolean Expressions.

- When a field of type **char** is used in a **fieldexp**, you may subscript the field name. For example,

```
bkmaster.agent[5,10]
```

  only uses the fifth through tenth characters of the **agent** field.

- Both constant expressions and **fieldexp**s can involve arithmetic operations. These are the available arithmetic operators:

Operator	**Operation**
+	addition
-	subtraction
*****	multiplicaton
/	division

  Multiplication and division have precedence over addition and subtraction. Where operators have equal precedence, the precedence is left to right. Use parentheses to override the default precedence rules.

- The keyword **rowid** can be used anywhere that a *fieldexp* can be used. **Rowid** represents the physical record number of a selected record. You may reference the **rowid** of a particular table with, for example: **bkmaster.rowid**.

- A date function can be used in expressions of the *selectlist*, *wcondition*, or *hcondition*. Each date function takes a single parameter which must be a **DATE** type expression. Where the date expression is an integer, this is interpreted as a number of days since December 31, 1899. These are the date functions and what they do:

day	returns the day of the month, 1-31.
month	returns the month of the year, 1-12.
weekday	returns the day of the week, 1-7.
year	returns the year, eg. 1988.

- The keyword variable **today** can be used anywhere that a constant can be used. **Today** equates to the current system date. Multiplication and division with **today** is inappropriate, but

```
today - bkmaster.bookdate
```

for example, will give use the number of days since **bookdate**.

- The keyword variable **user** is recognized on UNIX and XENIX systems and can be used anywhere that a constant can be used, though doing arithmetic with **user** is not appropriate. **User** equates to the current user's log-in name.

Select Clause

- To make a field name unambiguous, you may prefix the field with the table name, for example: **bkmaster.bkno**.
- The *selectlist* of ***** will select all the fields of all tables of the **from** clause. A *selectlist* expression of *table***.*** will cause all fields of *table* to be selected.
- You may add an alias to any constant expression or **fieldexp** in the *selectlist*. The expression and its alias are separated by a space. This alias is an identifier that would appear in the heading of the **select** output or would become a field name of *temptable* if there is an **into temp** clause.
- The results of selecting a date value is an expression with the format *mm*/*dd*/*yyyy*.
- If the *selectlist* contains an aggregate expression as well as a single record **fieldexp**, then there must be a **group by** clause and all table fields of the single record must be members of the *grouplist*.

From Clause

- You may add an alias to an table name in the *tablelist*. The table name and its alias are separated by a space. This allows you to access the same table twice with unique a way of specifying which access is being made:

```
select    itin.fromcity, cit1.name,
          itin.tocity, cit2.name
    from  itin, city cit1, city cit2
    where itin.fromcity = cit1.code and
          itin.tocity = cit2.code
```

- The **outer** keyword can be prepended to any table of the *tablelist*, except the first. The **outer** table must have a join relationship to a parent table. If there are no rows from the **outer** table satisfying the join, the qualifying rows of the parent or parents of the **outer** table will still be output in the **select** result. In a simple example we have the table fields **x.a** and **y.b** that contain these values:

x.a	**y.b**
1	2
2	

The following **select** statement:

```
select * from x, y
    where x.a = y.b
```

will produce:

```
2       2
```

However, if we add this **outer** relationship:

```
select * from x, outer y
    where x.a = y.b
```

the result will be:

```
1       -
2       2
```

Disjoint **outer** tables cannot be joined, such as tables **b** and **c** in:

```
select * from a, outer b, outer c
```

Here are some examples of hierarchical **outer** relationships:

```
select * from a, outer (b, outer c)

select * from a, outer (b, p, q, outer (c, y, z))
```

Where Clause

- The boolean expression of *wcondition* can involve comparisons with constant expressions as well as single record **fieldexp**s. There cannot be aggregate expressions in *wcondition*.
- To make a field name unambiguous, you may prefix the field with the table name.

Group By Clause

- All fields of the any single record *fieldexp*s in the *selectlist* must be included in the *grouplist*.
- The **group by** clause allows you to construct a *selectlist* containing both single record *fieldexp*s and aggregate *fieldexp*s. Without aggregate expressions in the *selectlist*, **select unique** will produce the same result as **group by**.
- Any aggregate expressions in the *selectlist* produce a separate result for each group.

Having Clause

- The boolean expression of *hcondition* can involve comparisons with constant expressions as well as aggregate properties of the group. There cannot be single record **fieldexp**s in *hcondition*. Without a **group by** clause, all the records selected comprise a single group.

Into Temp Clause

- *Temptable* exists only for as long as the SQLFLEX script that created it runs.
- The names of the fields in *temptable* are the names of the fields in the *selectlist*. Where an expression in the *selectlist* is other than a simple field name, in other words, a constant, an aggregate or date function, or any arithmetic expression, then the expression must be followed by a space and an alias name. This alias name will become the field name of the *temptable*.
- The keyword constant **null** can be an element of *selectlist* and is the way to initialize the corresponding field of *temptable* to null.

Examples

```
select agent, bookdate, salamount from bkmaster
    where salamount > 1000
    order by agent, bookdate desc;


select * from bkmaster, bkdetail
    where bkmaster.bookdate = bkdetail.bookdate
    into temp jointable;


select agent, sum( salamount) from bkmaster
    group by agent
    having sum( salamount) > 10000;
```

12.15 Boolean Expressions

Overview

A boolean expression evaluates to true or false.

There are seven different syntactical usages for the SQLFLEX simple boolean expression. These will be discussed under the following six subtopics.

Syntax

```
expr relop expr

fieldname matches "wildstr"

fieldname like "wildstr"

expr between constant and constant

expr in (valuelist)

expr in (selstmt)

expr selrelop { all | any | some } (selstmt)

exists (selstmt)

fieldname is null
```

Description

expr is a **fieldexp** as defined under the SELECT topic or a constant expression.

- The remaining syntax terms will be described under their related subtopic.

Notes

- These are the available relational operators:

Operator	Operation
=	equal
!= or **<>**	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

- Any number of simple boolean expressions can be combined with the logical operators **and** or **or** to make complex boolean expressions. The logical operator **and** has precedence over the operator **or**. Use parentheses to override the default precedence rule.
- Any boolean expression can be preceded by the boolean unary operator **not**. A complex boolean expression must be enclosed in parentheses to apply a **not** to it.
- For character data, uppercase letters have lesser value than lowercase letters. Thus, **Z** is less than **a**. Numbers are less than letters. The collating sequence of all letters follows the ASCII character set.
- For date expressions, greater than means later in time.
- The keyword identifiers **today** and **user** may be used for *expr*s of type **DATE** and **CHAR**, respectively.
- Date constants are quoted strings of the format *mm*/*dd*/*yy*.
- Only the last simple boolean expression of a complex boolean expression can have a *selstmt*.

SIMPLE COMPARISON

The simple comparison compares one expression to another using the standard boolean relational operators.

Syntax

expr relop expr

Description

expr	is a **fieldexp** as defined under the SELECT topic or a constant expression.
relop	is a relational operator.

Examples

```
status != "D"

agent = "Jones"  and  bookdate > today - 90
```

A table name prefix is only required in order to resolve ambiguities of like named fields in more than one table of the **select**'s **from** clause.

```
bkmaster.bookdate = bkmaster.postdate

bkmaster.comamount > 0.10 * bkmaster.salamount
```

This example uses the aggregate function **max**:

```
max( refamount) > 10000
```

WILDCARD STRING COMPARISON

The wildcard string comparison compares the value of a table field of type **CHAR** with a string pattern that contains possible wildcard characters.

Syntax

```
fieldname matches "wildstr"

fieldname like "wildstr"
```

Description

fieldname	is the name of a field of a table in the **select** statement's **from** clause.
matches	is a required keyword that specifies that a wildcard string comparison is to be done.
like	is a required keyword that specifies that a wildcard string comparison is to be done.
wildstr	is a string of characters that contains possible wildcard characters. The syntax of this wildcard string for the **matches** operator differs from that for the **like** operator.

Notes

- These are the available wildcards and how they are used for *wildstr* of the **matches** operator:

?	matches any single character.
*	matches zero or more characters.
[*c*…]	matches any of the characters enclosed in the brackets.
[*a*-*z*…]	matches any of the characters between *a* to *z*. There can be any number of the range specifications interspersed with individual characters within the brackets.
[^*c*…]	matches any of the characters not enclosed in the brackets. There may also may be range specifications interpersed with the characters. The **.** and ***** characters do not have wildcard meaning inside brackets.
\	removes any wildcard meaning of the next character. \ can also be used within **[ ]**'s.

- These are the available wildcards and how they are used for *wildstr* of the **like** operator:

_	matches any single character.
%	matches zero or more characters.

- *Fieldname* is a single field *fieldexp*, and therefore the wildcard string comparison cannot be used in the boolean expression of a **having** clause.

Examples

This example will match a three character pattern of the letter **A**, followed by any letter, followed by a digit between **0** and **2** or a **7**:

```
refno matches "A?[0-27]"
```

This example will match patterns not ending in **son**:

```
not agent matches "*son"
```

These are the corresponding examples using the **like** operator:

```
refno like "A_0" or refno like "A_1" or
refno like "A_2" or refno like "A_7"

not agent like "%son"
```

BETWEEN OPERATOR

The **between** operator allows you to test if an expression is in the inclusive range between two constant values.

Syntax

```
expr between constant and constant
```

Description

expr	is a **fieldexp** as defined under the SELECT topic.
between	is a required keyword.
constant	is a constant of the same data type as *expr*.
and	is a required keyword.

Notes

- Date constants are quoted strings of the format *mm*/*dd*/*yy*.
- The keyword identifiers **null**, **today**, and **user** may be used as values in *valuelist*, but not in ranges.

Example

```
bookdate between "01/01/90" and today
```

LIST COMPARISON

A list inclusion expression compares the value of an expression to one or more constant values in a list of values.

Syntax

```
expr in (valuelist)
```

Description

expr	is a **fieldexp** as defined under the SELECT topic or a constant expression.
in	is a required keyword.
valuelist	is a comma separated list of constants of the same data type as *expr*.

Notes

- *Valuelist* may include a range expression, which is two values separated by a dash, -. The first value of the range must be less than the second value. If the second value of the range is a negative number, enclose the number in parentheses.
- String constants in *valuelist* must be quoted.
- Date constants are quoted strings of the format *mm*/*dd*/*yy*.
- The keyword identifiers **null**, **today**, and **user** may be used as values in *valuelist*, but not in ranges.

Examples

```
state in ("CA", "NV", "AZ")
```

This is an example of a *valuelist* with ranges as well as individual values:

```
num in (-100-(-10), 10-100, 1000)
```

SUBQUERY COMPARISON

A subquery comparison compares an expression to the result of another **select** statement.

Syntax

```
expr in (selstmt)

expr selrelop { all | any | some } (selstmt)
```

Description

expr	is a **fieldexp** as defined under the SELECT topic or a constant expression.
in	is a required keyword.
selrelop	is a relational operator.
all	is a keyword that specifies a comparison test against all values returned by *selstmt*.
any	is a keyword that specifies a comparison test against any value returned by *selstmt*.
some	is a keyword that is synonymous with **any**.
selstmt	is a **select** statement.

Notes

- These are the available relational operators for the subquery comparison:

Operator	Operation
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

- The *selectlist* of the *selstmt* must be a single expression. It cannot be a comma separated list of expressions.
- If the *selstmt* does not return a value, then any comparison with **all (***selstmt***)** will return true.
- If the *selstmt* does not return a value, then any comparison with **any** or **some (***selstmt***)** will return false.
- The *selstmt* can have a condition in its **where** clause that depends on a value in the current record of the outer **select**. This is called a correlated subquery.
- The *selstmt* cannot contain an **order by** clause.

Example

```
bkdetail.bkno
    in (select bkno from bkmaster
              where agent = "Jones")


bkdetail.bkno >
    any (select bkno from bkmaster
              where agent = "Jones")


bkmaster.bookdate <=
    all (select bookdate from bkdetail
              where bkno = bkmaster.bkno)
```

SUBQUERY EXISTENCE

A subquery existence test determines if a **select** statement has any results at all.

Syntax

```
exists (selstmt)
```

Description

exists	is a required keyword.
selstmt	is a **select** statement.

Example

```
exists (select 1 from bkmaster
            where bookdate < "1/1/88")
```

Note that we are not interested in the value returned by the **select**, so we devise the simplest expression for the subquery's *selectlist*.

NULL TEST

The null test determines if the value of a table field is null.

Syntax

```
fieldname is null
```

Description

fieldname	is the name of a field of a table in the **select** statement's **from** clause.
is	is a required keyword.
null	is a required keyword.

Notes

- A field will have a null value if no value has ever been assigned to it or **null** has been assigned to it.
- *Fieldname* is a single field *fieldexp*, and therefore the null test cannot be used in the boolean expression of a **having** clause.

Example

```
bkmaster.bookdate is null
```

12.16 Aggregate Functions

Overview

An aggregate function derives its value from an operation across multiple records of a table.

These are the aggregate functions:

avg	returns the average for a numeric field.
count	returns the number of records.
max	returns the maximum value.
min	returns the minimum value.
sum	returns a sum for a numeric field.

Each aggregate function will be discussed under its separate subtopic.

AVG

The **avg** aggregate function returns the average of values from a numeric field.

Syntax

```
avg( field)
```

Description

avg	is a required keyword.
field	is a numeric field.

Notes

- Where there is a **group by** clause, **avg**, when used in a expression of the *selectlist*, returns the average for *field* from records selected for a group.
- Null values for *field* are not included in the average. If all values for *field* are null, then **avg(** *field***)** will return null.

Example

```
select agent, avg( salamount) from bkmaster
    group by agent
```

COUNT

The **count** aggregate function returns the number of records selected.

Syntax

```
count( * | distinct field)
```

Description

count	is a required keyword.
*	indicates that all records selected are to be counted.
distinct	is a keyword used when counting the number of unique values for a field.
field	is the field of the **distinct** clause.

Notes

- Where there is a **group by** clause, **count**, when used in a expression of the *selectlist*, returns the number of records selected for a group.

Examples

```
select count(*) from bkmaster;
```

```
select agent, sum( salamount) from bkmaster
    group by agent
    having count(*) > 100;
```

MAX

The **max** aggregate function returns the maximum value from a field.

Syntax

```
max( field)
```

Description

max	is a required keyword.
field	is a field.

Notes

- Where there is a **group by** clause, **max**, when used in a expression of the *selectlist*, returns the maximum value of *field* from records selected for a group.
- Where a field is of type **CHAR**, the maximum value is determined by the ASCII collating sequence.

Example

This example would provide the most recent booking date for each agent:

```
select agent, max( bookdate) from bkmaster
    group by agent
```

MIN

The **min** aggregate function returns the minimum value from a field.

Syntax

```
min( field)
```

Description

min	is a required keyword.
field	is a field.

Notes

- Where there is a **group by** clause, **min**, when used in a expression of the *selectlist*, returns the minimum value of *field* from records selected for a group.
- Where a field is of type **CHAR**, the minimum value is determined by the ASCII collating sequence.
- Null is considered the minimum value of any field.

Example

This example will select the alphabetical first agent:

```
select min( agent) from bkmaster;
```

SUM

The **sum** aggregate function returns the total of values from a numeric field.

Syntax

```
sum( field)
```

Description

sum	is a required keyword.
field	is a numeric field.

Notes

- Where there is a **group by** clause, **sum**, when used in a expression of the *selectlist*, returns the total of the values from *field* from records selected for a group.
- If all values for *field* are null, then **sum(** *field***)** will return null.

Example

```
select sum( salamount) from bkmaster;
```

12.17 UNLOAD

Overview

The **unload** statement writes selected data from a database table to a specially formatted ASCII text file. This is useful in transfering information between computers or databases in a machine independent format.

Syntax

```
unload to "unloadfile" selectstmt
```

Description

unload to	are required keywords.
unloadfile	is the required absolute or relative path name of a text file that will contain the data that is unloaded.
selectstmt	a **select** statement used to extract the database records for *unloadfile*.

Notes

- Each line of *unloadfile* will contain the data of a single row result of the **select** statement. The data of each field of *unloadfile* is terminated with the vertical bar character, | .
- *Selectstmt* may be any legal **select** statement, but may not use an **into temp** clause.

Example

```
unload to "alltypes.unld"
    select * from alltypes;
```

12.18 UPDATE

Overview

The **update** statement updates one or more records of database table.

Syntax

```
update table
     set      field = expr
          [ , field = expr ] . . .
     [ where wcondition ]
```

Description

update	is a required keyword.
table	is the name of the table for which records will be updated.
set	is a required keyword.
field	is a field of *table*.
expr	is a constant expression or a field expression.
where	is an optional keyword for a clause that specifies selection criteria that may reduce the scope of the records updated.
wcondition	is the boolean expression of the **where** clause.

Notes

- An **update** statement without a **where** clause will update all records of *table*.
- The syntax for *wcondition* is exactly the same as for the **select** statement.
- Almost any expression that can be used to make up a *selectlist* expression of a **select** statement can be used in *expr*. The exceptions are ***** and aggregates.
- The **update** statement locks *table* from other users' modifications during the update operation.
- Decimal precision loss or string truncation may occur and is reported as warning messages when it does. The warning messaging can be turned off with the **-nw** command line option to **fxsql**.
- A **select** statement used in an **update** cannot have an **into temp** or **order by** clause.
- Serial type fields cannot be updated.

Examples

```
update alltypes
     set fchar2 = "XY",
         fsmallint = fsmallint + 1,
         fdate = today
     where fmoney < 10000;
```

13. TERMINAL SETUP

Overview

This chapter shows how to manage your terminals using the **Terminal Control File** The **Terminal Contol File** is required in order to run Infoflex screens under UNIX operating systems only. The **Terminal Control File** is where you will define your terminal's command sequences for such characteristics as video attributes and key recognition.

Terminal Control File

This section describes how to install and edit a terminal control file. The terminal control file contains command sequences that are specific to your terminal model. These command sequences control terminal characteristics such as video attributes and key recognition. The following 2 subsections describe how to Install and Edit the **Terminal Control File**.

Installing the Terminal Control File

Install the **Terminal Control File** by assigning the control file name to the environment variable **FXTERM**. Below is the command to set **FXTERM**.

FXTERM=wyse60; export FXTERM

Normally you will not need to set **FXTERM** because it defaults to the value of **TERM**.

Editing the Terminal Control File

If the **Terminal Control File** must be modified or does not exist, you will need to edit/create it. To edit/create a terminal control file, select **Terminal Control** on the **Development Menu** or enter the following command. Before entering the command be sure to set the infoflex environment variables as per chapter 2.

flexterm

The following menu will appear:

```
SELECT TERMINAL TYPE

 1. tvi910   : TeleVideo 910
 2. tvi920   : TeleVideo 912C/920C
 3. tvi925   : TeleVideo 925
 4. vt52     : DEC VT52
 5. adm3a    : LSI ADM 3A
 6. viewpt   : ADDS Viewpoint/3A Plus
 7. altos3   : Altos III
 8. altos4   : Altos IV
 9. altos5   : Altos V
10. wyse     : Wyse WY-100
11. wyse50   : Wyse 50+
12. wyse60   : Wyse 60
13. pcunx    : ISC UNIX 5.3
14. pcxnx    : SCO XENIX
15. ansi      : SCO XENIX V/386

Enter terminal number (a=add, q=quit):  8
```

If the terminal is not on the menu, add it by entering an 'a' to the above prompt. The system will prompt you for the terminal name and then add it to the list of selectable terminals.

After selecting a terminal the following menu will appear:

```
T E R M I N A L   S U P P O R T   M E N U (/usr2/fx/dev)
-------------------------------------------------------

 1.  Assign CRT & ACTION KEY Settings.
 2.  Assign ACTION KEYS Settings via Keyboard.
 3.  Display CRT Settings.
 4.  Display ACTION KEY Settings.
 5.  Print Settings.
 6.  Test Settings.
 7.  Save Settings.
 8.  Save & Install Settings to /usr2/fx/dev.
 9.  Load Settings from TERMCAP file.
 Q.  Quit.

     Select Menu Option #
```

The following subtopics discuss what each above menu option does.

1. Assign CRT & ACTION KEY Settings

You will be placed in an editor to assign control sequences. The control sequences are written in **termcap** style (eg., the **ESC** character is **\E**).

2. Assign ACTION KEYS Settings via Keyboard

In this mode, you can define the control sequence for a action key by simply pressing the action key at the keyboard. Action keys are defined as non-data entry keys such as Function keys, Arrow keys, and Control keys.

3. Display CRT Settings

Terminal output control sequences will be displayed at the terminal. A ***** by the capability description indicates that the capability is required by Infoflex. A ****** indicates that the capability is desirable.

4. Display ACTION KEY Settings

Keyboard input control sequences for action keys will be displayed at the terminal.

5. Print Settings

Control sequences are printed.

6. Test Settings

This option is not implemented at this time.

5. Save settings

This option saves the control sequence settings to the directory **.../fx/src/term.**

6. Save & Install settings to .../fx/dev

This option saves the control sequence settings to the directories **.../fx/src/term** and **.../fx/dev**. The version in **.../fx/dev** will be used by Infoflex at run-time.

9. Load Settings from TERMCAP file

This option loads the control sequences definitions from your UNIX **/etc/termcap** file. Be sure that the terminal name assigned in the **termflex.dir** file is the same as the one in **termcap**.

After loading from **termcap**, the **A_TYPE** capability may need to be updated. **A_TYPE** tells TERMFLEX how to set a video attribute. This flag is set to **1** if the terminal sets attribute bytes before and after an output string. The flag is **0** if the terminal does not bracket the output string with attribute bytes. This setting corresponds to the **sg#** parameter of **termcap**. TERMFLEX assumes that the terminal will use a consistent method for outputting video attributes. This may not always be the case as **termcap** will allow a mixture of methods (eg.: **sg#0**, **ug#1**). To resolve this you will want to use the specific attribute control sequences in your terminal manual to define attribute settings.

Q. Quit

The TERMFLEX program will terminate.

Defining a New Terminal Capability

This section describes how to define a new terminal capability. In order to use this section you must have the Infoflex-4GL source code.

The steps for defining a new terminal capability are as follows

1) Define a new macro name in the **fxcrt.h** header file in **$FXDIR/include** that represents the capability. For example,

```
#define  NEWMACRO  5000
```

2) Add an entry into the **termflex.crt** file located in the **$FXDIR/src/term** directory. This file defines each of the macros and what they represent.

3) If your new macro was NOT appended to the end of the macro list in **fxcrt.h** then you will need to do the following:

Recompile the entire Infoflex system using the **flexmake -r** command. The Programmer's Guide describes this process.

Also, you will need to reinstall each terminal capability file. To do this, run termflex menu selection #8 for each terminal type.

14. PRINTER SETUP

Overview

This chapter shows how to manage your printers using the **Printer Control File** and the **Printer Configuration File** (options 18 and 19 on the System Administration menu). The **Printer Control File** is where you will define your printer's command sequences for compressed print and pitch. The **Printer Configuration File** is where you will specify additional parameters about each printer such as its identification name, printer control file name, character width, and page length. Both of these printer files are optional and are only necessary if you require the control features offered by them.

Printer Control File

This section describes how to install and edit a printer control file. The printer control file contains command sequences that are specific to your printer model. These command sequences control printer characteristics such as compressed print and pitch levels. The following 2 subsections describe how to Install and Edit the **Printer Control File**.

Installing the Printer Control File

Install the **Printer Control File** by assigning the control file name to the environment variable **FXPRINT**. Below is the command to set **FXPRINT**.

FXPRINT=hplaser; export FXPRINT

Editing the Printer Control File

If the **Printer Control File** must be modified or does not exist, you will need to edit/create it. To edit/create a printer control file, select **Printer Control** on the **Development Menu** or enter the following command. Before entering the command be sure to set the infoflex environment variables as per chapter 2.

flexprnt

The following menu will appear:

```
SELECT PRINTER TYPE

 1. tosh351   : Toshiba P351/P321/P341
 2. hplaser2  : HP Laserjet Series II
 3. ex800     : Epson EX-800
 4. pan1091   : Panasonic KX-P1091
 5. nec2080   : Nec 2080

Enter printer number (a=add, q=quit):
```

If the printer is not on the menu, add it by entering an 'a' to the above prompt. The system will prompt you for the printer name and then add it to the list of selectable printers.

After selecting a printer the following menu will appear:

```
P R I N T E R   S U P P O R T   M E N U (/usr2/fx/dev)
-----------------------------------------------------

    1.  Assign settings.
    2.  Display settings.
    3.  Print settings.
    4.  Test settings.
    5.  Save settings.
    6.  Save & Install settings to /usr2/fx/dev.
    Q.  Quit.

        Select Menu Option #
```

The following subtopics briefly describe what each above menu option does.

1. Assign settings

You will be placed in an editor to assign control sequences. The control sequences are written in **termcap** style (eg., the **ESC** character is **\E**).

2. Display settings

Control sequences will be displayed at the terminal.

3. Print settings

Control sequences are printed.

4. Test settings

This option is not implemented at this time.

5. Save settings

This option saves the control sequence settings to the directory **.../fx/src/term.**

6. Save & Install settings to .../fx/dev

This option saves the control sequence settings to the directories **.../fx/src/term** and **.../fx/dev**. The version in **.../fx/dev** will be used by Infoflex at run-time.

Q. Quit

The flexprnt program will terminate.

Printer Configuration File

This section describes how to install and edit a **Printer Configuration File**. The **Printer Configuration File** allows you to specify additional parameters about each printer such as its identification name, printer control file name, character width, and page length. The following 2 subsections describe how to Install and Edit the **Printer Configuration File**.

Installing the Printer Configuration File

Installation of the **Printer Configuration File** consists of assigning the environment variable **FXPRT** the fullpath of where the configuration file resides. A sample configuration file can be found in the directory **.../fx/dev/prconfig**.

To set the **FXPRT** environment variable for the sample configuration file, enter the following command.

FXPRT=.../fx/dev/prconfig; export FXPRT

Note that you should copy the sample **prconfig** file to a private area so any future updates will not overwrite your changes.

Editing the Printer Configuration File

The next step after installing your printer configuration file is to customize it for your site.

To customize the configuration file, select **Printer Configuration** on the **Development Menu** or enter the following command. Before entering the command be sure to set the infoflex environment variables as per chapter 2.

flexprc

Upon entering this command, you will be placed in an editor in order to modify the configuration file.

The sample configuration will appear as follows.

```
Infoflex OS       Printflex              Bottom End
Name     Name     Name      Width Length Margin Feed Options
-------- -------- --------- ----- ------ ------ ---- --------
0        laserjet  hplaser  80    60      2      Y
1        laserjet  hplaser  80    60     10      Y  -olandscape
2        laserjet2 hplaser 170    60
3        deskjet   hplaser 170    60
disk     Serial    hplaser  80    60
```

Below is a description of each field or column.

Infoflex Name

This is the name Infoflex uses to refer to the printer. To route an Infoflex report to this printer you would enter a **Report Destination** of **P** followed by the Infoflex printer name. For example,

Report Destination: P1

If you enter a **P** without an Infoflex printer name, it will default to the Infoflex printer name **0**.

Note that users can have different default printers by assigning them different **Printer Configuration Files**.

OS Name

This is the operating system's destination name for this printer. For Windows/NT use lpt1, lpt2, etc..

Printflex Name
This is the name of the **Printer Control File** that applies to this printer.

Width
This is the printer's character width. If the report output exceeds this width it will automatically be compressed (provided the compressed print sequences are defined in the **Printer Control File**).

Length
This is the printer's lines per page. This is important for correctly aligning pages.

Bottom Margin
Not used at this time.

End Formfeed
Enter "Y" if you would like a formfeed at the end of each report.

Options
These options are passed *as is* to the printers interface program. One popular option is the landscape option which would be specified here as *-olandscape*. This feature is not available on DOS/WINDOWS.

Defining a New Printer Capability

This section describes how to define a new printer capability. In order to use this section you must have the Infoflex-4GL source code.

The steps for defining a new printer capability are as follows

1) Define a new macro name in the **fxcrt.h** header file in **$FXDIR/include** that represents the capability. For example,

```
#define   NEWMACRO   5000
```

2) Add an entry into the **prntflex.prt** file located in the **$FXDIR/src/term** directory. This file defines each of the macros and what they represent.

3) If your new macro was NOT appended to the end of the macro list in **fxcrt.h** then you will need to do the following:

Recompile the entire Infoflex system using the **flexmake -r** command. The Programmer's Guide describes this process.

Also, you will need to reinstall each terminal capability file. To do this, run termflex menu selection #8 for each terminal type.

APPENDIX

Sample SCREENFLEX Program with INSTRUCTIONS

```
TABLES
  slmord
  sldord
  tbcust
  inven
END


SELECT

     slmord( invno)
       EXTRACTALL
     sldord ( invno) slmord( invno)
       EXTRACTALL

END


SCREEN  slinv   aaftersave( aaftsave) aafterdelete( aaftdelete)
{
ACCOUNTFLEX    [modemsg      ]     INVOICE ENTRY SCREEN       DATE:[today   ]
================================================================================
Invoice#:[invno  ] Customer:[custno|custname          ] Invoice Date:[entdate ]
        Qty/             Item                            Unit       Extended
Units x Unit  = Qty      Code    Description             Price      Price
-----   -----   -------  ------  ----------------------  ----------  ----------
[unit ][qunit][qor     ][ai    ][adesc                ][uprice    ][eprice    ]









================================================================================
                                                          Total:[totsale    ]
}
END

ATTRIBUTES

modemsg = displayonly type character, noupdate, noentry, reverse, retain;
today = displayonly type date, default=today, noupdate, noentry;

invno = slmord.invno, required, searchby(slmord.slminvno),
        beforedit( befinvno);

custno = slmord.custno, upshift, required, right, truncate,
         searchby( slmord.slmcustno),
   lookup( tbcust.tbcustkey, slinv.custname = tbcust.name),
   tablehelp( "flex tbcusts asdc", tbcust.tbcustname, tbcust.code, tbcust.name);
custname = displayonly type character, noupdate, noentry,
   comments = "Enter Client Name";

entdate= slmord.entrydate, required,  afterfield( aftentdate),
         comments = "Enter Entry Date";

REPEAT(11)

unit = sldord.unit_ordered, default="1", afteredit(calcext);
qunit = sldord.qty_perunit, default="1", afteredit(calcext);
qor  = sldord.qty_ordered, default = "1", required,
       format="###", noupdate, noentry;
ai = sldord.inven_no, upshift, required, afterfield(ai_calcext),
   lookup( inven.invenkey, slinv.adesc = inven.description),
   tablehelp( "flex invens asdc", inven.invenkey, inven.code, inven.description);
adesc = displayonly type character, noupdate, noentry;
uprice = sldord.unit_price, format="##.##", afteredit( calcext);
eprice = sldord.ext_price, format="##.##",
         total( slinv.totsale);

ENDREPEAT
```

```
totsale = slmord.totsale, format="#,###.##", noupdate, noentry;

END


INSTRUCTIONS


static int calcflag = NO;
static long gentdate;

aaftsave()
{

  if ( calcflag) {
     dmaprec( @slmord, @slinv);
     if ( fmrewcurr( @slmord) < 0)
        msggerr( "!Unable to save slmord file");
     calcflag = NO;
     }

  return(0);
}


aaftdelete()
{
   if (flexmode == ADDMODE)
      return(0);

   dmaprec( @slmord, @slinv );
   fmrewcurr( @slmord);
    return(0);
}


ai_calcext()
{
    if ( isempty( @slinv.uprice) == YES) {
       $slinv.uprice = $inven.sale_price;
       tmapfld( @slinv.uprice);
       }

    calcext();
    return(0);
}



calcext()
{
    SLIST *pslist;

     calcflag = YES;

     $slinv.qor = $slinv.unit * $slinv.qunit;
     tmapfld( @slinv.qor);
     $slinv.totsale = $slinv.totsale - $slinv.eprice;
     $slinv.eprice  = $slinv.unit      * $slinv.uprice;
     $slinv.totsale = $slinv.totsale + $slinv.eprice;
     tmapfld( @slinv.eprice);
     tmapfld( @slinv.totsale);
     return(0);
}

befinvno()
{
     if ( flexmode != ADDMODE)
        return(0);

     $slinv.entdate = gentdate;

     if ( isempty( @slinv.entdate) == YES)  {
        gentdate = curdate;
        $slinv.entdate = gentdate;
        }

    tmapfld( @slinv.entdate);

      /* so screen not saved unless other fields edited */
    setmodfld( @slinv.entdate, OFF);


    return(0);
}


aftentdate()
{
```

```
    gentdate = $slinv.entdate;

    return(0);
}

END
```

Sample SCREENFLEX Program with MAIN function

```
TABLES
  apmctl  (open read)
  apbinv (open)
  apminv (open)
  apdinv (open)
  apminv (alias armalias open)
  apbopen (open)
  apmven (open)
  glmcoa
END


SELECT
     apbinv( module, source, batch)
       EXTRACTALL
END



SCREEN  apinv  frame window(0,0,4,79)  beforesubsection( bef1subsection)
               beforedelete( bef1delete)
{

 ACCOUNTFLEX    [modemsg      ]   A/P Invoice Batch Entry     DATE:[today   ]

 Source:<[m]-[s]> Batch:<[batch ]> Entry Date:[batdate ]     Total:[batchtotal]
}
END

ATTRIBUTES

modemsg = displayonly type character,noupdate, noentry, reverse, retain;
today = displayonly type date, default=today, noupdate, noentry, retain;
m(module)     = apbinv.module, default=apmctl.module, noupdate, noentry;
s(source)     = apbinv.source, default="I", noupdate, noentry;
batch = apbinv.batch,
 comments = "Enter Batch Number then press SAVE Function Key to enter Invoices";
batdate = apbinv.entdate, noupdate, default=today, afteredit( ae_batdate),
 comments = "Press SAVE Function Key to enter Invoices";
batchtotal = apbinv.amount, noupdate, noentry;

END


SELECT
  apminv(module, source, batch, invno, venno)
    EXTRACTALL
  apdinv (module, source, batch, invno, venno) apminv( module, source, batch, invno, venno)
   EXTRACTALL
END


SCREEN  apdist box window(5, 0)  joinon( apbinv.apbinvbatch)
          beforesubsection( befsubsection)
          beforerow( befrow)
          beforesave( befsave)
          afterdelete(aftdelete)

          abeforerow( abefrow)
          aaftersave( aafsave)
          aafterdelete(aafdelete)
{
                [modemsg      ]                                  [batchtotal]
 Invoice:<[invno     ]> Vendor:<[venno ]>[venname              ]
 Invoice Date:[trandate] Period Date:[perdate ] [pdiv     |div]
 Discount Date:[discdate] Discount Allowed:[discallow   ]
 Due Date:[duedate ] Desc:[description                          ]
 @_____________________________________________________________________________
 Account-#      Account Title                                      Amount
 @_____________________________________________________________________________
 [glcode      ][gldesc                       ]                  [amount     ]



 @_____________________________________________________________________________
  [m][s] [batch  ]                                        Total:[totamount   ]
}
END

ATTRIBUTES
modemsg = displayonly type character,noupdate, noentry, reverse, retain,
          setrow(-4);
m(module)     = apminv.module, noentry, noupdate, nodisplay;
```

```
s(source)     = apminv.source, noentry, noupdate, nodisplay;
batch = apminv.batch, nodisplay, noentry, noupdate, nodisplay;
batchtotal = apbinv.amount, noupdate, noentry, retain, setrow( -2);

invno = apminv.invno,  upshift, truncate, required, formatfield( vinvformat),
        beforedit( be_autoinvno), afterfield( af_autoinvno)
        searchby(apminv.apminvinvno);
venno = apminv.venno, upshift,  truncate, required, formatfield( cusformat),
   searchby( apminv.apminvvennoinvno),
   lookup( apmven.apmvenvenno, apdist.venname = apmven.name),
   tablehelp( "", apmven.apmvenname, apmven.venno, apmven.name),
   comments="Enter Vendor Code (press HELP key to see list)";
venname = displayonly type character, noupdate, noentry;

trandate= apminv.trandate, required, default = today, afteredit( ae_trandate),
         comments = "Enter Transaction Date";
perdate = apminv.perioddate, required, default = today,  afteredit( ae_perdate),
         comments = "Enter Fiscal Period Date";
pdiv    = displayonly type character, noupdate, noentry, retain;
div     = apminv.divno, formatfield( divformat);
discdate = apminv.discountdate;
discallow = apminv.discountallow, format="#,###.##";
duedate = apminv.duedate;
description = apminv.description;


REPEAT(5)

glcode = apdinv.glcode, required, formatfield( glformat),
    lookup( glmcoa.glmcoaglcode, apdist.gldesc = glmcoa.description),
    tablehelp("", glmcoa.glmcoaglcode, glmcoa.glcode, glmcoa.type, glmcoa.description),
   comments="Enter G/L Account Code (press HELP key to see list)";
gldesc = glmcoa.description, noupdate, noentry;
amount = apdinv.amount, format="##.##", total( apdist.totamount);

ENDREPEAT

totamount = apminv.amount, format="#,###.##", noupdate, noentry;

END


INSTRUCTIONS

static double savetotamount;

main()
{
   static char modearg[6];

 for ( ; ; ) {
   if ( ESCAPEKEY == flexcmd( "flex apinv ACD NNNNNYY") )
      break;
   strcpy( modearg, "ACD");
   if ( compare( @apbinv.module, @apmctl.module) != 0  ||
         isempty( @apbinv.postno) == NO)
       strcpy( modearg, "V-V");
   for ( ; ; ) {
        sprintf( msgbuf, "flex apinv -f apdist %s", modearg);
        if ( ESCAPEKEY == flexcmd( msgbuf ) )
           break;
        }
   smaprec( @apbinv, @apinv);
   }

   clrscr();
}


/*****************************************************************************
               A P I N V
*****************************************************************************/
static int bef1subsection()
{
    popsavekey( 7, 16, "Press SAVE function key to enter Invoices");
    return(0);
}
/*
static int bef1row()
{
    if ( compare( @apbinv.module, @apmctl.module) == NO ||
         isempty( @apbinv.postno) == NO)
        altermode(  VIEWMODE);
    else
        altermode( CHANGEMODE);
    return(0);
}
*/

static int bef1delete()
```

```
{
    if ( flexmode == CHANGEMODE )  {
        if (*($apmctl.module) != *($apinv.module) ) {
            sprintf( msgbuf, "Batch may NOT be deleted because from different source (%s)",
                 $apinv.module);
            msggerr( msgbuf);
            return(-1);
            }
        move( @apbinv.module, @apminv.module);
        move( @apbinv.source, @apminv.source);
        move( @apbinv.batch, @apminv.batch);
        if (0 > delchildren( @apminv, @apminv.apminvbatch,
                  @apminv.batch, NULLFUNC) )
           return(-1);
        }

  return(0);
}

static int ae_batdate()
{
    return(-2);
}


/*****************************************************************************
               A P D I S T  - Header
*****************************************************************************/
static int befsubsection()
{
   move( @apbinv.amount, @apdist.batchtotal);
   tmapfld( @apdist.batchtotal);
   divprompt( $apmctl.divexist, @apdist.pdiv);
   return(0);
}


static int befrow()
{
   if (flexmode == ADDMODE) {
     if ( 0 > chkmdup() )
        return(-1);
     }

     return(0);
}


static int aftdelete()
{
   updbatch( $apdist.totamount * -1);
   return(0);
}



static int befsave()
{
  int rtn;
  char sinvno[20];


  if ( 0 > chkmdup() )
      return(-1);

  move( @apbinv.batch, @apdist.batch );
  move( @apbinv.batch, @apminv.batch );
  tmapfld( @apdist.batch);
  if (flexmode == ADDMODE && isautoinvno( @apdist.invno) ) {
            rtn = getinvoice( "apmctl", "invoice", "inv_prefix", sinvno);
            putdf( sinvno, @apminv.invno);
            putsf( sinvno, @apdist.invno);
            tmapfld( @apdist.invno);
            /*
            sprintf( msgbuf, " unique %s", sinvno);
            msggerr( msgbuf);
            */
            return( rtn);
       } /* end of ADDMODE */


   return(0);
}

/*****************************************************************************
               A P D I S T  - Array
*****************************************************************************/
static int abefrow()
{
   savetotamount = $apdist.totamount;
```

```
return(0);
}


static int aafsave()
{
   updbatch( $apdist.totamount - savetotamount);
   return(0);
}

static int aafdelete()
{
   updbatch( $apdist.totamount - savetotamount);
   return(0);
}

static int updbatch( diffamount)
double diffamount;
{

   if (diffamount == 0.0)
      return;

  /*
   sprintf( msgbuf, "updatebatch: %lf", diffamount);
   msggerr( msgbuf);
 */

   $apbinv.amount += diffamount;
   if ( 0 > fmrewcurr( @apbinv) )  {
      fmerrmsg( @apbinv);
      }
   move( @apbinv.amount, @apdist.batchtotal);
   tmapfld( @apdist.batchtotal);

/*
   sprintf( msgbuf, "New total: %lf", $apbinv.amount);
   msggerr( msgbuf);
*/
}


static int ae_trandate()
{
   if (chkrdate( $apdist.trandate, $apmctl.daytolerance,
                $apmctl.datereference, $apmctl.datemin, $apmctl.datemax) < 0)
       return(-1);
   move( @apdist.trandate, @apdist.perdate);
   tmapfld( @apdist.perdate);
   return(0);
}

static int ae_perdate()
{
   return( chkrdate( $apdist.perdate, $apmctl.daytolerance,
                     $apmctl.datereference, $apmctl.datemin, $apmctl.datemax));
}


static int be_autoinvno()
{
   return( beautoinvno( @apmctl.invoice, @apdist.invno) );
}

static int af_autoinvno()
{
   return( afautoinvno( @apmctl.invoice, @apdist.invno) );
}

static int chkmdup()
{

     if (isautoinvno( @apdist.invno) )
        return(0);

 if (flexmode == ADDMODE) {
    move( @apdist.venno, @armalias.venno);
    move( @apdist.invno, @armalias.invno);
    if (fmstart(@armalias, @armalias.apminvvennoinvno, 0, ISEQUAL) == 0 &&
        fmread( @armalias, ISCURR) == 0)  {
          sprintf(msgbuf,"@Invoice No. currently assigned in batch %ld",
                    $armalias.batch);
          msggerr(msgbuf);
          return(-1);
          } /* end of find */
     } /* ADDMODE */

    move( @apdist.venno, @apbopen.venno);
    move( @apdist.invno, @apbopen.invno);
    if (fmstart(@apbopen, @apbopen.apbopenvennoinvno, 16, ISEQUAL) == 0 &&
```

```
        fmread( @apbopen, ISCURR) == 0)  {
       if (*($apbopen.source) != 'C') {
          msggerr( "@WARNING: Invoice No. already exists");
          return(-1);
          }
       return(0);
       }

    return(0);
}


END
```

Sample SCREENFLEX Program using ZOOM

```
TABLES
  apmctl (read)
  apmven
  apmven (alias apmvenalias)
  apstmnt
  apbopen
  apmopen
  glmcoa
  pomship
  slmtax
  slmsale
  slmterm
END

SELECT
   apmven(venno)
       EXTRACTALL
   apbopen( venno, trandate, invno) apmven( venno)
       EXTRACTALL
END

SCREEN  apveni frame   azoomkey( azoomdetail) user1key( agekey)
                       beforesubsection( befsubsection)
{

 ACCOUNTFLEX   [modemsg      ]    Vendor Inquiry Screen      DATE:[today   ]

 Vendor Code:[venno ][name                          ]
 Contact:[contact                       ] Tel:[phone          ]
 Terms:[tc][termdesc            ] Credit Code:[cred] Credit Limit:[credlimit  ]
 @_____________________________________________________________________________
  Invoice    Date        Description                                  Amount Due
 @_____________________________________________________________________________
 [invno     |trandate] [description                             ][amountdue   ]




 @_____________________________________________________________________________
 Last Post Date:[postdate][trandat2]                  Total Due:[curbalance   ]
       Last Date         MTD  Last Month         YTD     Last Year    Pending
 Order:[orddate |orders_mtd |orders_lm  |orders_ytd  |orders_lyr  |orders_pend]
 Purch:[saledate|purch_mtd  |sales_lm   |purch_ytd   |sales_lyr   ]
}
END


ATTRIBUTES

modemsg = displayonly type character, noupdate, noentry, reverse, retain;
today = displayonly type date, noupdate, noentry, default=today, retain;

venno= apmven.venno, upshift, truncate, formatfield( cusformat), required,
       searchby(apmven.apmvenvenno),
   tablehelp( "", apmvenalias.apmvenname, apmvenalias.venno, apmvenalias.name),
    comments = "Enter the Vendor's code (Press HELP key for list)";
name = apmven.name,  searchby( apmven.apmvenname),
       comments = "Enter Vendor's name";
contact = apmven.contact;
phone = apmven.phone, phone,  comments = "Enter Phone number";
tc    = apmven.termscode, upshift,
        lookup( slmterm.slmtermtermcode, apveni.termdesc = slmterm.description),
        tablehelp( " ",slmterm.slmtermtermcode,
                slmterm.termcode,slmterm.description),
       comments="Enter Terms Code for this vendor";
termdesc = displayonly type character, noupdate, noentry;
cred    = apmven.credit_code, upshift;
credlimit  = apmven.credit_limit, format="#,###,###.##";

REPEAT(5)
invno = apbopen.invno, upshift, truncate,  formatfield( vinvformat);
trandate= apbopen.trandate, required,
         comments = "Enter Invoice Date";
description = apbopen.description;
amountdue=apbopen.amountdue, format="#,###.##";
ENDREPEAT

curbalance  = apmven.current_balance, format="#,###,###.##";
postdate    = apmven.latest_postdate;
trandat2    = apmven.latest_trandate;
orddate     = apmven.orders_lastdate;
orders_mtd  = apmven.orders_mtd, format="#,###.##";
orders_lm   =apmven.orders_prevmonth, format="#,###.##";
orders_ytd  = apmven.orders_ytd, format="#,###.##";
orders_lyr  =apmven.orders_prevyear, format="#,###.##";
```

```
orders_pend = apmven.orders_pending, format="#,###.##";
saledate    = apmven.purch_lastdate;
purch_mtd   = apmven.purch_mtd, format="#,###.##";
sales_lm    =apmven.purch_prevmonth, format="#,###.##";
purch_ytd   = apmven.purch_ytd, format="#,###.##";
sales_lyr   = apmven.purch_prevyear, format="#,###.##";

END



SELECT
     apbopen( venno, invno, trandate)
      EXTRACT
     apmopen ( venno, invno, trandate, tranrecno) apbopen( venno, invno)
       EXTRACTALL
END


SCREEN  apopen frame
{

 ACCOUNTFLEX    [modemsg      ]      Vendor Inquiry         Date:[today   ]

 Vendor:[venno |venname               ] Invoice:[invno     ] Date:[trandate]
 Source:[m][s] Batch:[batch  ] Desc:[description                           ]
 @_____________________________________________________________________________
  Date      Source Batch  Description                    Discount      Amount
 @_____________________________________________________________________________
 [atrandat][M|S|BATCH |adescription                    |adiscount|aamount    ]




 @_____________________________________________________________________________
     Invoices  + Adjustments  -    Discounts  -      Checks =          Due
 [invamount  ] + [amountadj]  - [discountpay] - [amountpay  ] = [amountdue  ]

}
END

ATTRIBUTES

modemsg = displayonly type character,noupdate, noentry, reverse, retain;
today = displayonly type date, default=today, noupdate, noentry, retain;

venno = apbopen.venno, upshift, truncate, required, formatfield( cusformat),
   beforedit( be_venno),
   searchby( apbopen.apbopenvennoinvno),
   lookup( apmven.apmvenvenno, apopen.venname = apmven.name),
   tablehelp( "", apmven.apmvenname, apmven.venno, apmven.name);
venname = displayonly type character, noupdate, noentry;
invno = apbopen.invno, upshift, truncate,  formatfield( vinvformat),
        searchby(apbopen.apbopeninvno);
trandate= apbopen.trandate, required,
         comments = "Enter Invoice Date";

m      = apbopen.module;
s      = apbopen.source;
batch  = apbopen.batch;

description = apbopen.description;


REPEAT(6)

atrandat= apmopen.trandate, required,
         comments = "Enter Entry Date";
M       = apmopen.module;
S       = apmopen.source;
BATCH   = apmopen.batch;
adescription = apmopen.description;
adiscount    = apmopen.discountpay, format="#,###.##";
aamount = apmopen.amount, format="#,###.##";

ENDREPEAT

invamount = apbopen.amount, format="#,###.##";
amountadj = apbopen.amountadj, format="#,###.##";
amountpay   = apbopen.amountpay, format="#,###.##";
discountpay   = apbopen.discountpay, format="#,###.##";
amountdue     = apbopen.amountdue, format="#,###.##";

END


SCREEN agescreen popup frame  window( 6, 3)  beforesubsection( befagesub)
{
```

```
                    A G E D   B A L A N C E

    Balance [a0         |a1         |a2         |a3         |a4         ]
[balamount  |age0       |age1       |age2       |age3       |age4       ]

                   Press ESCAPEKEY to exit  [w]
}
END

ATTRIBUTES
a0  = displayonly type character, noupdate, noentry;
a1  = displayonly type character, noupdate, noentry;
a2  = displayonly type character, noupdate, noentry;
a3  = displayonly type character, noupdate, noentry;
a4  = displayonly type character, noupdate, noentry;

balamount   = displayonly type money, format="##,###.##", noupdate, noentry;
age0        = displayonly type money, format="b##,###.##", noupdate, noentry;
age1        = displayonly type money, format="b##,###.##", noupdate, noentry;
age2        = displayonly type money, format="b##,###.##", noupdate, noentry;
age3        = displayonly type money, format="b##,###.##", noupdate, noentry;
age4        = displayonly type money, format="b##,###.##", noupdate, noentry;
w           = displayonly type character;
END


INSTRUCTIONS

#include "act.h"

static long  m_recno;
static long  trandate;


static int azoomdetail()
{
    move( @apbopen.trandate, &trandate);
    flexcmd( "flex apveni -f apopen vs-vs");
    return(0);
}


static be_venno()
{
   flexkey = SAVEKEY;
   move( @apveni.venno,    @apopen.venno);
   move( @apveni.invno,    @apopen.invno);
   move( &trandate,        @apopen.trandate);
   tmaprec( @apopen);

   return(-1);
}

static int befsubsection()
{
    keychglabel(USER1KEY, "AGE ");
    return(0);
}

static agekey()
{
    flexcmd( "flex apveni -f agescreen  P");
    wpage( @apveni, NO);
    funcmflag = NO;
    return(0);
}

static int befagesub()
{
    double totamount = 0.0;

    agevendor();
    totamount += $agescreen.age0 + $agescreen.age1;
    totamount += $agescreen.age2 + $agescreen.age3;
    totamount += $agescreen.age4;
    move(&totamount,  @agescreen.balamount);
    tmaprec( @agescreen);
    return(0);
}

#define ARCUSI
static char agesort[5];
static char agetype[5];
#include "apread.flx"

END
```

Sample SCREENFLEX Program for PURGING

```
TABLES
   slmaster
   sldetail
   sltrans
END

SCREEN  select
{
T.A.M.S                    PURGE SALES RECORD BY DEPART DATE     DATE:[today    ]
================================================================================

       This Purging program will delete all bookings where the Booking Number
       and Departure Date are both less than or equal to the ones entered below:

       Before Purging do the following:
             1) Backup all data to tape and save perminently.
             2) Be sure all Reciepts and Payments have been POSTED.
             3) Make sure all bookings prior to the Booking Number and
                Departure Date entered below are PAID !!!


           Booking No: [bkno    ]     Departure Date: [depart  ]


       After Purging do the following:
             1) Run Reports for UNapplied Payments to see if all is OK.

================================================================================
}
END

ATTRIBUTES

today = displayonly type date, default = today,  noupdate, noentry;
bkno   = displayonly type integer, required;
depart = displayonly type date, required;
END


INSTRUCTIONS

static FILE *fplog;

main( argc, argv)
int argc;
char *argv[];
{

   flexcmd( "flex slpurge -f select p");
   if ( flexkey == ESCAPEKEY)
      fxabort(0);

   logopen( argv[0]);

   loghead( argv[0]);

   doprocess( NO);

   logclose();

   fxabort(0);
}



doprocess( updateflag)
int updateflag;
{
   int mode = ISINOUT+ISEXCLLOCK;
   int count;
   long bkno = 0L;
   long departdate;


   postopen( @slmaster, @slmaster.slmbkno,  mode);
   postopen( @sldetail, @sldetail.sldbkno,  mode);
   postopen( @sltrans,  @sltrans.sltbkno,   mode);

   /* clear joined fields */
   for ( ; ; )  {

      /************** Find NEXT MASTER Record **************/
      $slmaster.bkno = bkno;
      if ( 0 > fmfind( @slmaster, @slmaster.slmbkno, ISGREAT)  ) {
          if ( iserrno != 110 && iserrno != 111) {
```

```
            fmmsgbuf( msgbuf, @slmaster);
            logmsg( msgbuf);
            break;
            }
         break;
         }
      bkno = $slmaster.bkno;

      if ( bkno > $select.bkno)
         break;

     /************** What's Latest Departure Date **************/
       $sldetail.bkno = $slmaster.bkno;
       $sldetail.recno = 0L;
       mode = ISGTEQ;
       departdate = 0L;
       for ( count = 0; ; ++count)  {

            /************ Find all matching DETAIL record ******************/
           if ( 0 > fmfind( @sldetail, @sldetail.sldbkno, mode)  ) {
              if ( iserrno != 111 && iserrno != 110) {
                  fmmsgbuf( msgbuf, @sldetail);
                  logmsg( msgbuf);
                  }
              break;
              }

           mode = ISNEXT;

           /* if detail's join field(s) have changed then ...*/
           if ( compare( @slmaster.bkno, @sldetail.bkno) )
              break;

           if ($sldetail.departdate > departdate)
              departdate = $sldetail.departdate;

           }  /* end of Detail loop */

       if ( departdate > $select.depart)
           continue;

       dopurge( bkno, departdate);


       }  /* end of Master loop */

    /* close CISAM files  */
   fmclose( @slmaster);
   fmclose( @sldetail);
   fmclose( @sltrans);

   return(0);
}


dopurge( bkno, departdate)
long bkno;
long departdate;
{
    char sdate[10];
    long recno;

    datestr( departdate, sdate, "MM/DD/YY");             0, bkno, sdate);
    sprintf( msgbuf, "Purging slmaster Booking#: %ld - %s
    logmsg( msgbuf);

    if (0 > fmdelcurr(@slmaster) ) {
       fmmsgbuf( msgbuf, @slmaster);
       logmsg( msgbuf);
       fxabort(-1);
       }

    /*** DELETE SLDETAIL ***/
    recno = 0L;
    for ( ;;)  {
           $sldetail.bkno = bkno;
           $sldetail.recno = recno;
           if ( 0 > fmfind( @sldetail, @sldetail.sldbkno, ISGREAT)  ) {
               if ( iserrno != 111 && iserrno != 110) {
                   fmmsgbuf( msgbuf, @sldetail);
                   logmsg( msgbuf);
                   }
               break;
               }

           if ( compare( &bkno, @sldetail.bkno) )
               break;
                                                                  0,
           sprintf( msgbuf, "Purging sldetail Booking#: %ld - %ld
                 $sldetail.bkno, $sldetail.recno);
```

```
            logmsg( msgbuf);
            if (0 > fmdelcurr(@sldetail) ) {
               fmmsgbuf( msgbuf, @sldetail);
               logmsg( msgbuf);
               }
            recno = $sldetail.recno;
            }  /* end of Detail loop */

    /*** DELETE SLTRANS ***/
    dclrrec( @sltrans);
    recno = 0L;
    for ( ;;)  {
            $sltrans.bkno = bkno;
            $sltrans.voucher = recno;
            if ( 0 > fmfind( @sltrans, @sltrans.sltbkno, ISGREAT)  ) {
                if ( iserrno != 111 && iserrno != 110) {
                    fmmsgbuf( msgbuf, @sltrans);
                    logmsg( msgbuf);
                    }
                break;
                }

            if ( compare( &bkno, @sltrans.bkno) )
                break;
                                                                        0,
            sprintf( msgbuf, "Purging sltrans Booking#: %ld - %ld - %ld
                   $sltrans.bkno, $sltrans.voucher, $sltrans.recno);
            logmsg( msgbuf);
            if (0 > fmdelcurr(@sltrans) ) {
               fmmsgbuf( msgbuf, @sltrans);
               logmsg( msgbuf);
               }
            recno = $sltrans.voucher;
            }  /* end of Detail loop */


 }




loghead( filename)
char *filename;
{
                        0, filename);
  sprintf( msgbuf, "%s:
  logmsg( msgbuf);                  0);
  logmsg( "Booking   Departure Date                                      0);
  logmsg( "--------- ------------------------- ----------- -----------

}


logopen( logname)
char *logname;
{
   char filename[20];

   strcpy( filename, logname);
   strcat( filename, ".log");

    if ( NULL == (fplog = fopen(filename, "w")) ) {
         sprintf( msgbuf, "error opening log file %s0, filename);
         logmsg( msgbuf);
         return( -1);
         }

}

logclose()
{
    fclose( fplog);
}

logmsg( postbuf)
char *postbuf;
{
     fprintf( fplog, "%s", postbuf);
     printf( "%s", postbuf);

}

postopen( pdbhead, pdbindex, mode)
DBHEAD *pdbhead;
DBINDEX *pdbindex;
int mode;
{

  /* Open File for posting  - */
```

```
    if ( 0 > fmopen( pdbhead, mode) ) {
       if ( iserrno == 107) {
           sprintf( msgbuf, "File (%s) already in use - try again later",
               getrtxt( pdbhead->dhName));
           msggerr( msgbuf);
           fxabort(0);
           }
       fmmsgbuf(msgbuf, pdbhead);
       logmsg( msgbuf);
       fxabort(-1);
       }

    if ( mode & ISOUTPUT)
       return;

    if ( 0 > fmfind( pdbhead, pdbindex, ISFIRST) ) {
       if ( iserrno != 110) {
           fmmsgbuf(msgbuf, pdbhead);
           logmsg( msgbuf);
           fxabort(-1);
           }
       }

    fmread( pdbhead, ISPREV);

}

END
```

Sample REPORTFLEX Program

```
TABLES
  apmctl  (read)
  apbinv
  apminv
  apdinv
  apmven
  glmcoa
END


SCREEN  select frame
{

 ACCOUNTFLEX              A/P Invoice Batch Listing            DATE:[today   ]

       Report Destination:[d         ] (S=Screen, Pn=Printer, Dn=Disk, A=Aux)
       Report Copies:     [c ]   (1 - 10)
       Report Title Page: [t]    (Y=Yes, N=No)
       Report Detail:     [n]    (Y=Yes, N=No)

       Source:       [m]

       Batch Range:  [batch1  ]    to  [batch2  ]

       Invoice Range:[invno1    ]  to  [invno2    ]

       Vendor Range:   [venno1]      to  [venno2]
                         [venname1      ][venname2        ]

}
END

ATTRIBUTES

today = displayonly type date, default = today,  noupdate, noentry;
d(rptdest)   = displayonly type character, required, upshift;
c(rptcopies) = displayonly type smallint,  required, include( 1 to 10);
t(rpttitle)  = displayonly type character, required, upshift, include( Y, N);
n(rptnoprint) = displayonly type character, required, upshift, default="Y",
               include( Y, N);

m(module) = apminv.module, upshift,
  comments="Enter Source of Invoices to print (R=A/P,S=Sales, =ALL)";

batch1 = apminv.batch,
   lookup( apbinv.apbinvbatch),
   tablehelp( "", apbinv.apbinvbatch, apbinv.batch, apbinv.entdate, apbinv.amount),
   comments="Enter Batch Number begin range";
batch2 = apminv.batch,
   lookup( apbinv.apbinvbatch),
   tablehelp( "", apbinv.apbinvbatch, apbinv.batch, apbinv.entdate, apbinv.amount),
   comments="Enter Batch Number end range";

invno1 = apminv.invno, upshift, truncate, formatfield( vinvformat),
   lookup( apminv.apminvinvno),
   tablehelp( "",apminv.apminvinvno, apminv.invno, apminv.venno, apminv.amount),
   comments="Enter Invoice begin range to print";
invno2 = apminv.invno, upshift, truncate, formatfield( vinvformat),
   lookup( apminv.apminvinvno),
   tablehelp( "",apminv.apminvinvno, apminv.invno, apminv.venno, apminv.amount),
   comments="Enter Invoice end range to print";

venno1 = apminv.venno, upshift, truncate, formatfield( cusformat),
  lookup( apmven.apmvenvenno, select.vename1 = apmven.name),
  tablehelp("",apminv.apminvvennoinvno, apminv.venno, apminv.invno,apminv.amount),
  comments="Enter Vendor Code begin range to print (press HELP key for list)";
venno2 = apminv.venno, upshift, truncate, formatfield( cusformat),
  lookup( apmven.apmvenvenno, select.vename2 = apmven.name),
  tablehelp("",apminv.apminvvennoinvno, apminv.venno, apminv.invno,apminv.amount),
   comments="Enter Vendor Code end range to print (press HELP key for list)";

venname1 = displayonly type character, noupdate, noentry;
venname2 = displayonly type character, noupdate, noentry;
END



SELECT

     apminv( module, source, batch, invno, venno)
       EXTRACTALL
     apdinv( module, source, batch, invno, venno)
                              apminv( module, source, batch, invno, venno)
```

```
      EXTRACTALL
    apbinv( module, source, batch) apminv( module, source, batch)
      EXTRACT
    apmven( venno)  apminv( venno)
      EXTRACT
    glmcoa( glcode)  apdinv( glcode)
      EXTRACT
    WHERE whereselect

END


REPORT

heading breakon( apminv.batch)  newpage everypage
{
ACCOUNTFLEX            A/P Invoice Batch Listing              Page:[page    ]
                                                          Date:[today   ][time    ]
================================================================================
Source:<[m]-[s]> Batch:[batch  ] Batch Date:[batdate ]    Total:[batchtotal    ]

--------------------------------------------------------------------------------
Invoice-#  Vendor Name            Invoice  Period   Due                Amount
--------------------------------------------------------------------------------
}

heading breakon( apminv.venno) afterprint( aftmpostmsg)
{
[invno     ][venno |venname       ][trandate|perdate |duedate ] [mamount       ]
       [div|description                            ][discdate|discallow  ]
}

heading breakon( apminv.venno) rptnoprint
{
                     Account-#    Description                           Amount
}

detail  rptnoprint afterprint( aftdpostmsg)
{
                    [glcode      |gldesc                       |amount        ]
}

total  breakon( apminv.venno) rptnoprint
{
                                                 ------------------------------
                                                 Invoice Total:  [amount       ]

}

total  breakon( apminv.batch)
{
                                                 ==============================
                                                 Batch Total:   [amount        ]
}

total
{



                                                  ****************************
                                                  ****************************
                                                  Grand Total:  [amount        ]
}
END

ATTRIBUTES

today     = displayonly type date, default=today;
time      = displayonly type mtime, default=time;
page      = displayonly type smallint;
m         = apbinv.module;
s         = apbinv.source;
batch     = apbinv.batch;
batdate   = apbinv.entdate;
batchtotal= apbinv.amount, format="#,##,##.##";
invno     = apminv.invno, formatfield( vinvformat);
venno     = apminv.venno, formatfield( cusformat);
venname   = apmven.name;
trandate  = apminv.trandate;
perdate   = apminv.perioddate;
div       = apminv.divno;
duedate   = apminv.duedate;
discdate   = apminv.discountdate;
discallow   = apminv.discountallow;
description = apminv.description;
mamount    = apminv.amount, format="#,##.##";

glcode    = apdinv.glcode, formatfield( glformat);
gldesc    = glmcoa.description;
```

```
amount    = apdinv.amount, format="#,##.##";

END

INSTRUCTIONS


static int aftmpostmsg()
{
    if ( isempty( @apminv.postno) == NO)
       rptline( "* * *       P O S T E D       * * *");
    return(0);
}

static int aftdpostmsg()
{
    if ( isempty( @apdinv.postno) == NO)
       rptline( "* * *       P O S T E D       * * *");
    return(0);
}
END
```

Sample REPORTFLEX Program for CHECK Printing

```
TABLES
   apmctl  (read)
   apbchk
   apmchk
   aptchk
   apbopen
   apmven
   apmreg
   apcheck
   glmcoa
END

SCREEN  select frame
{

 ACCOUNTFLEX                Print Checks                        DATE:[today    ]

     @_____________________________________________________________________
     | This program will print Checks for any batch of payments.  Below    |
     | enter the Batch and Check number range to assign. These Check       |
     | Numbers will automatically be assigned to all payments without      |
     | Check Numbers.  You may also enter a range of Check Numbers to VOID |
     | should checks become destroyed by aligning printer or paper jams.   |
     |                                                                     |
     |                 Once You have made your selections                  |
     |             Press the SAVEKEY to start or ESCAPEKEY to exit         |
     _____________________________________________________________________
       Report Destination:[d         ] (S=Screen, Pn=Printer, Dn=Disk, A=Aux)
       Source:[m]
       Batch: [batch1  ] G/L:[glcode1     |glcodedesc1                      ]

       Print Check Number Range:[chknofirst]  to  [chknolast ]

       VOID  Check Number Range:[chknov1   ]  to  [chknov2   ]
}
END

ATTRIBUTES

today = displayonly type date, default = today,  noupdate, noentry;
d(rptdest)   = displayonly type character, required, upshift,
               default=apmctl.chkprinter;

m  = apmchk.module, upshift, default=apmctl.module, required,
  comments="Enter Source of Checks to print (P=A/P,O=Orders)";

batch1 = apmchk.batch, required,
   tablehelp( "", apbchk.apbchkbatch, apbchk.batch, apbchk.entdate, apbchk.amount),
   comments="Enter Batch Number (Press the HELP key to list batches)";

glcode1= displayonly type character, formatfield( glformat), noupdate, noentry,
    lookup(glmcoa.glmcoaglcode, select.glcodedesc1 = glmcoa.description),
    tablehelp("flex glmcoa asdc", glmcoa.glmcoaglcode,
                glmcoa.glcode, glmcoa.type, glmcoa.description);

glcodedesc1 = displayonly type character, noupdate, noentry;


chknofirst = displayonly type integer, right,
   comments="Enter First Check Number to print";

chknolast = displayonly type integer, right,
   comments="Enter Last Check Number to print";

chknov1 = displayonly type character, right, truncate,
   comments="Enter First Check Number to VOID";

chknov2 = displayonly type character, right, truncate,
   comments="Enter Last Check Number to VOID";

END


SELECT  section( chkroutine)

     apcheck(recno)
          EXTRACTALL

END

REPORT

heading breakon(apcheck.recno) newpage everypage
                            pagelength(42) beforeprint( bp_heading)
{
```

```
                                                                [chkno     ]

                                                              [chkdate ]

          [vendor                          ]                  [chkamt     ]

     [chkamountword                                                          ]

        [vendor                        ] [venno ]
        [address1                      ]
        [address2                      ]
        [address3                      |zip       ]



}

detail  printlength(37)
{
   [trandate] [invno     ]   [amount    ]     [discountpay]   [amountpay]
}

total breakon(apcheck.recno) printline(39)
{
   [chkdate ] [chkno     ]   [amount    ]     [discountpay]   [amountpay]
}

END


ATTRIBUTES

chkamountword = displayonly type character;
chkdate  = apcheck.chkdate;
chkno    = apcheck.chkno;
chkamt   = apcheck.chkamount;
venno    = apcheck.venno;
vendor   = apmven.name;
address1 = apmven.address1;
address2 = apmven.address2;
address3 = apmven.address3;
zip      = apmven.zip;

trandate     = apcheck.invdate;
invno        = apcheck.invno;
amount       = displayonly type money, format="#,###.##";
discountpay  = apcheck.discountpay, format="#,###.##";
amountpay    = apcheck.amountpay, format="#,###.##";
END

INSTRUCTIONS

#include "act.h"

static int voidflag;

static int bp_heading()
{
   char   amountword[100];

   if (voidflag) {
       move("**************** V O I D ***********************",@heading.chkamountword);
       $heading.chkamt = 0.0;
       }
   else {
       moneytowrds($apmchk.amount, amountword, sizeof(amountword), 1);
       move (amountword, @heading.chkamountword);
       }
   return(0);
}


moneytowrds(value, wordvalue, maxlen, flag)
double   value;
char     *wordvalue;
int      maxlen;
int      flag;
{
    int  i;
    int  arydollar[5];
    char wordcents[3];
    long longvalue;
    long cents, dollars;
    static char *singles[] =
                  { "", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
                  "Eight", "Nine" };
    static char *teens[] =
                  { "", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen",
                    "Sixteen", "Seventeen", "Eighteen", "Nineteen" };
    static char *tens[] =
```

```
                { "", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty",
                "Seventy", "Eighty", "Ninety" };

        longvalue = value;
        dollars = longvalue / 100;
        for (i=(sizeof(arydollar)/sizeof(int)); i > 0; i--) {
           arydollar[i-1] = dollars % 10;
           dollars /= 10;
           }

        wordvalue[0] = ' ';
        if (arydollar[0] != 0 || arydollar[1] != 0) {
             if ( arydollar[0] == 0 && arydollar[1] != 0)
                      strcat(wordvalue,singles[arydollar[1]]);
             else if (arydollar[0] > 0 && arydollar[1] == 0)
                      strcat(wordvalue, tens[arydollar[0]]);
             else if (arydollar[0] == 1 && arydollar[1] > 0)
                      strcat(wordvalue, teens[arydollar[1]]);
             else if (arydollar[0] > 1 && arydollar[1]  > 0) {
                      strcat(wordvalue,tens[arydollar[0]]);
                    strcat(wordvalue, " ");
                      strcat(wordvalue,singles[arydollar[1]]);
                      }
            strcat(wordvalue, " Thousand ");
            }

         if  (arydollar[2] != 0)  {
             strcat(wordvalue, singles[arydollar[2]] );
             strcat(wordvalue, " ");
             strcat(wordvalue, "Hundred");
             strcat(wordvalue, " ");
           }

         if (arydollar[3] != 0 || arydollar[4] != 0) {
             if (arydollar[3] > 0 && arydollar[4] == 0)
                      strcat(wordvalue, tens[arydollar[3]]);
             else if (arydollar[3] == 0 && arydollar[4] > 0)
                      strcat(wordvalue, singles[arydollar[4]]);
             else if (arydollar[3] == 1 && arydollar[4] > 0)
                      strcpy(wordvalue, teens[arydollar[4]]);
             else if (arydollar[3] > 1  && arydollar[4] > 0) {
                      strcat(wordvalue, tens[arydollar[3]]);
                    strcat(wordvalue, " ");
                      strcat(wordvalue, singles[arydollar[4]]);
                      }
           strcat(wordvalue, " ");
           }

         strcat(wordvalue, "and ");
        cents   = longvalue % 100;
        ultoa( cents, wordcents);
         strcat(wordvalue, wordcents);
         strcat(wordvalue,"/100");

         if (flag) {
          for (i = strlen( wordvalue); i < maxlen; i++)
              wordvalue[i] = '*';
          }

}


static int trancount;
extern double getunapplied();

static int chkroutine()
{
    int rtn;

    fmclose( @apmchk); /* Opened by SELECT */

    rtn = doprocess();

    /* close all files in case error occurs in lower level function */
    fmclose( @apbchk);  /* One record per batch of checks */
    fmclose( @apmchk);  /* One record per check  */
    fmclose( @aptchk);  /* One record per Invoice paid by check */
    fmclose( @apmreg);  /* Check Register file (one record per posted check) */
    fmclose( @apmven);   /* Vendor file */
    fmclose( @apbopen);  /* Open Invoice File */

    return(rtn);
}



static int doprocess()
{

    if (isempty( @select.chknofirst) == NO) {
```

```
        if ( isempty( @select.chknolast) == NO)
            sprintf( msgbuf,
           "Are you sure you want to PRINT Checks %ld thru %ld(Y/N or Esc)?",
              $select.chknofirst, $select.chknolast);
        else
            sprintf( msgbuf,
           "Are you sure you want to PRINT Checks starting with %ld(Y/N or Esc)?",
              $select.chknofirst);
        if (chkprompt( msgbuf) < 0)
           return(-1);
        if (chkprint(1, "Verify Printing") < 0)
           return(-1);
        if (trancount > 0) {
           if (chkprint(2, "Printing") < 0)
              return(-1);
           }
        }

     return(0);
}



chkprint( passflag, passname)
int passflag;
char *passname;
{
    int m_mode, d_mode;
    long chkno;
    char char_chkno[20];
    double chkamount;
    long save_pageno;
    long rptpageno();
    double unapplied;

    trancount = 0;

    if (fmopen( @apbchk, ISINOUT+ISEXCLLOCK) < 0) {
       fmerrmsg(@apbchk);
       msggerr( msgbuf);
       return(-1);
       }

    if (fmstart( @apbchk, @apbchk.apbchkbatch, 0, ISFIRST) < 0) {
       if (iserrno != 110) {
          fmerrmsg(@apmchk);
          msggerr( msgbuf);
          return( -1);
          }
       return(0);
       }

    if (fmopen( @apmchk, ISINOUT+ISEXCLLOCK) < 0) {
       fmerrmsg(@apmchk);
       msggerr( msgbuf);
       return(-1);
       }
    if (fmstart( @apmchk, @apmchk.apmchkbatch, 0, ISFIRST) < 0) {
       if (iserrno != 110) {
          fmerrmsg(@apmchk);
          msggerr( msgbuf);
          return( -1);
          }
       return(0);
       }

    if (fmopen( @aptchk, ISINOUT+ISEXCLLOCK) < 0) {
       fmerrmsg(@aptchk);
       msggerr( msgbuf);
       return(-1);
       }
    if (fmstart( @aptchk, @aptchk.aptchkbatch, 0, ISFIRST) < 0) {
       if (iserrno != 110) {
          fmerrmsg(@aptchk);
          msggerr( msgbuf);
          return( -1);
          }
       }

    if (fmopen( @apmreg, ISINOUT+ISEXCLLOCK) < 0) {
       fmerrmsg(@apmreg);
       msggerr( msgbuf);
       return(-1);
       }

    if (fmopen( @apmven, ISINPUT+ISMANULOCK) < 0) {
       fmerrmsg(@apmven);
       msggerr( msgbuf);
       return(-1);
       }
```

```
if (fmstart( @apmven, @apmven.apmvenvenno, 0, ISFIRST) < 0) {
   if (iserrno != 110) {
      fmerrmsg(@apmven);
      msggerr( msgbuf);
      return( -1);
      }
   }

if (fmopen( @apbopen, ISINPUT+ISMANULOCK) < 0) {
   fmerrmsg(@apbopen);
   msggerr( msgbuf);
   return(-1);
   }
if (fmstart( @apbopen, @apbopen.apbopeninvno, 0, ISFIRST) < 0) {
   if (iserrno != 110) {
      fmerrmsg(@apbopen);
      msggerr( msgbuf);
      return( -1);
      }
   }

dclrrec( @apbchk);
move( @select.m,        @apbchk.module);
move( "C",              @apbchk.source);
move( @select.batch1,   @apbchk.batch);
if (fmread( @apbchk, ISEQUAL) < 0) {
   if (iserrno != 110 && iserrno != 111) {
      fmerrmsg(@apbchk);
      msggerr( msgbuf);
      return( -1);
      }
   sprintf( msgbuf, "@Invalid Batch: %s-ld", $apbchk.module, $apbchk.batch);
   msggerr( msgbuf);
   return(-1);
   }

chkno = $select.chknofirst;
dclrrec( @apmchk);
move(@apbchk.module,      @apmchk.module);
move(@apbchk.source,      @apmchk.source);
move(@apbchk.batch,       @apmchk.batch);
for (m_mode = ISGTEQ;;m_mode = ISNEXT) {
    if (fmread( @apmchk, m_mode) < 0) {
        if (iserrno != 110 && iserrno != 111) {
           fmerrmsg(@apmchk);
           msggerr( msgbuf);
           return( -1);
           }
        break;
        }

    if ( compare( @apmchk.module, @apbchk.module) != 0)
       break;
    if ( compare( @apmchk.source, @apbchk.source) != 0)
       break;
    if ( compare( @apmchk.batch, @apbchk.batch) != 0)
       break;
    if (isempty( @apmchk.chkno) == NO)
        break;

    move( @apmchk.venno, @apmven.venno);
    if (fmread( @apmven, ISEQUAL) < 0) {
       if (iserrno != 110 && iserrno != 111) {
          fmerrmsg(@apmven);
          msggerr( msgbuf);
          return(-1);
          }
       sprintf( msgbuf, "@Invalid Vendor Code: %s", $apmchk.venno);
       msggerr( msgbuf);
       return(-1);
       }

    chktoa( chkno, char_chkno);
    move( char_chkno, @apmchk.chkno);
    chkamount = 0.0;
    voidflag = NO;
    dclrrec( @aptchk);
    move( @apmchk.module,      @aptchk.module);
    move( @apmchk.source,      @aptchk.source);
    move( @apmchk.batch,       @aptchk.batch);
    move( @apmchk.recno,       @aptchk.m_recno);
    for (d_mode=ISGTEQ; *($apmven.method_flag) == 'O';d_mode=ISNEXT) {

        if (fmread( @aptchk, d_mode) < 0) {
           if (iserrno != 110 && iserrno != 111) {
              fmerrmsg(@aptchk);
              msggerr( msgbuf);
              return( -1);
              }
           break;
```

```
        }

    if ( compare( @aptchk.module, @apmchk.module) != 0)
       break;
    if ( compare( @aptchk.source, @apmchk.source) != 0)
       break;
    if ( compare( @aptchk.batch, @apmchk.batch) != 0)
       break;
    if ( compare( @aptchk.m_recno, @apmchk.recno) != 0)
       break;

    move( @aptchk.invno, @apbopen.invno);
    move( @apmchk.venno, @apbopen.venno);
    if (fmread( @apbopen, ISEQUAL) < 0) {
       if (iserrno != 110 && iserrno != 111) {
          fmerrmsg(@apbopen);
          msggerr( msgbuf);
          return(-1);
          }
       sprintf( msgbuf, "@Invalid Invoice (%s-%s)",
              $aptchk.invno, $apmchk.venno);
       msggerr( msgbuf);
       return(-1);
       }

    chkamount +=  $aptchk.amountpay;
    if (passflag == 2) {
       save_pageno = rptpageno();
       output();
       if (save_pageno != rptpageno() ) {
          dclrrec( @apmreg);
          move(@aptchk.module,    @apmreg.module);
          move(@aptchk.source,    @apmreg.source);
          move(@aptchk.batch,     @apmreg.batch);
          move(@apbchk.glcode,    @apmreg.glcode);
          move(@apmchk.venno,     @apmreg.venno);
          move("V",               @apmreg.status);
          chktoa( chkno,          char_chkno);
          move(char_chkno,        @apmreg.chkno);
          move(&curdate,          @apmreg.trandate);
          $apmreg.amount = 0.0;
          if (0 > fmwrite( @apmreg)) {
             if ( iserrno != 100) {
                fmerrmsg( msgbuf, @apmreg);
                return(-1);
                }
             /* must be VOID check already because recno = 0 */
             sprintf(msgbuf,"@Check No(%s) is already VOID",
                   $apmchk.chkno);
             msggerr( msgbuf);
             }
         ++chkno;
         }

       } /* passflag == 2 */

    voidflag = YES;
    } /* end of aptchk loop */

++trancount;
if (passflag == 2 && *($apmven.method_flag) == 'B') {
   dclrrec( @apbopen);
   dclrrec( @aptchk);
   move( "BALFWRD",           @aptchk.invno);
   move( @apmchk.amount,      @aptchk.amountpay);
   output();
   }

if (*($apmven.method_flag) == 'O') {  /* Open Item Vendor */
    unapplied = getunapplied();
    if (mnyround( unapplied) != 0.0) {
       sprintf( msgbuf, "@Check Amount (%8.2lf) NOT fully applied",
                $apmchk.amountpay);
       msggerr( msgbuf);
       return(-1);
       }
    if (mnyround( $apmchk.amountpay) != mnyround( chkamount) ) {
       sprintf( msgbuf,
       "@Check Amount (%8.2lf) does NOT equal Invoice Amounts (%8.2lf)",
            $apmchk.amountpay, chkamount);
       msggerr( msgbuf);
       return(-1);
       }
    }

if (passflag == 2) {
   if (fmrewcurr( @apmchk) < 0) {
      fmerrmsg(@apmchk);
      msggerr( msgbuf);
      return(-1);
```

```
            }
         }


        sprintf( msgbuf, "%s Check No: %s", passname, $apmchk.chkno);
        msgstat( msgbuf);

        ++chkno;
        if (isempty( @select.chknolast) == NO &&
              chkno > $select.chknolast)
           break;
        }

    fmclose( @apbchk);
    fmclose( @apmchk);
    fmclose( @aptchk);
    fmclose( @apmreg);
    fmclose( @apmven);
    fmclose( @apbopen);
    return(0);
}

static int output()
{

    dclrrec( @apcheck);
    move( @apmchk.recno, @apcheck.recno);
    move( @apmchk.venno, @apcheck.venno);
    move( @apmchk.chkno, @apcheck.chkno);
    move( @apmchk.trandate, @apcheck.chkdate);
    move( @apmchk.amount, @apcheck.chkamount);
    move( @apbopen.trandate, @apcheck.invdate);
    move( @aptchk.invno,     @apcheck.invno);
    move( @aptchk.amountpay, @apcheck.amountpay);
    move( @aptchk.discountpay, @apcheck.discountpay);

     rptprint();
}


chktoa( chkno, char_chkno)
long chkno;
char *char_chkno;
{
    char tempbuf[20];
    int length;

    ultoa( chkno, tempbuf);

    /* right justify into 10 byte field */

   length = strlen( tempbuf);
   memset( char_chkno, ' ', 10);
   strcpy( &char_chkno[10 - length], tempbuf);
   char_chkno[10] = ' ';
}


static int chkprompt(msg)
char *msg;
{
   int keyint;

   msgstat( msg);
   keyint = getkey();
   if ( keyint != 'Y' && keyint != 'y')
       return(-1);
   if ( keyint == ESCAPEKEY)
       return(-1);
    return(0);
}

static double getunapplied()
{
    double unapplied = 0.0;

   if (*($apmctl.module) == 'O' || *($apmctl.module) == 'P') {
      if (*($apmven.method_flag) == 'O')
          unapplied = $apmchk.amount  - $apmchk.amountgl  - $apmchk.amountpay;
      else
          $apmchk.amountpay = $apmchk.amount  -  $apmchk.amountgl;
      }
   else {
      if (*($apmven.method_flag) == 'O')
          unapplied = $apmchk.amount  + $apmchk.amountgl  - $apmchk.amountpay;
      else
          $apmchk.amountpay = $apmchk.amount  +  $apmchk.amountgl;
        }

    return( unapplied);
```

```
}
END
```

Sample ISAMFLEX Program using dynamic file access

```
#include "flex.h"

          /******  SQL commands used to create the 'tbref' file
                create table tbref
                    (code          char(6),
                     name          char(15)
                     );
                create unique index tbrefkey
                        on tbref   (code);
          ******/

typedef struct {
     char      code[6 + 1];
     char      name[15 + 1];
     } TBREF;

TBREF tbref;

          /******  SQL commands used to create the 'tbemp' file
              create table tbemp
                  (code          char(4),
                   lname         char(15),
                   fname         char(10),
                   hire_date     date,
                   socno         char(13),
                   raise_date    date,
                   paymethod     integer,
                   salary        money,
                   commission    float
                   );
             create unique index tbempkey
                   on tbemp   (code);
          ******/

struct dbview empview[] =
    {
    {"code"},
    {"lname"},
    {"fname"},
    {"hire_date"},
    {"salary"}
    };


typedef struct {
         char  code[4 + 1];
         char  lname[15 + 1];
         char  fname[10 + 1];
         long  hire_date;
         double salary;
     } TBEMP;

TBEMP tbemp;


main()
{
    char tempdate[10];

      fmload("tbref tbemp");

      if (fmopen( getdhp("tbref"), ISINOUT+ISMANULOCK) < 0) {
         fmerrmsg( getdhp("tbref"));
         exit(1);
         }

      fmstructview( getdhp("tbref"), NULLCHAR, 0, (char *)&tbref);

      for ( ; ; ) {

          if (fmread( getdhp("tbref"), ISNEXT) < 0) {
             if (iserrno == 110) /* end-of-file */
                 break;
             fmerrmsg( getdhp("tbref"));
             exit(1);
             }
                                          0, tbref.code, tbref.name);
         printf( "TBREF: code=%s, name=%s

         }

      if (fmclose( getdhp("tbref")) < 0) {
         fmerrmsg( getdhp("tbref"));
         exit(1);
         }
```

```
    /* Use DBVIEW method for assingning data base fields to structures */

      if (fmopen( getdhp("tbemp"), ISINOUT+ISMANULOCK) < 0) {
         fmerrmsg( getdhp("tbemp") );
         exit(1);
         }

      fmstructview( getdhp("tbemp"), empview,
             sizeof( empview)/sizeof(struct dbview), (char *)&tbemp);

      for ( ; ; ) {

          if (fmread( getdhp("tbemp"), ISNEXT) < 0) {
             if (iserrno == 110) /* end-of-file */
                 break;
             fmerrmsg( getdhp("tbemp"));
             exit(1);
             }
                                                0,
         printf( "TBEMP: code=%s, name=%s, %s
                         tbemp.code, tbemp.lname, tbemp.fname);

         datestr( tbemp.hire_date, tempdate, NULLCHAR);
         printf( "       hire=%s, salary=%6.2lf
                  tempdate, tbemp.salary/100);

         }

      if (fmclose( getdhp("tbemp")) < 0) {
         fmerrmsg( getdhp("tbemp"));
         exit(1);
         }

     exit(0);
}
```

ERRORS

Compiler Errors

When compiling Infoflex source files using the **fxpp** command, errors may result from using incorrect syntax or invalid environment variables. Upon detecting an error, Infoflex will display a descriptive error message along with the source line number responsible. Error messages resulting from an inablity to access the data base, may also display an error code. The following topic contains a table describing these error codes.

Runtime Error

When running an Infoflex program using the **flex** command, errors may occur as a result of invalid data or invalid environment variables. Upon detecting an error, Infoflex will display a descriptive error message on the bottom line of your screen. These messages are contained in the file **sysmsg.flx** which is located in the application **bin** directory. **Sysmsg.flx** is a text file that you may edit in order to customize your messages. Error messages resulting from an inablity to access the data base, may also display an error code. These error codes are as follows:

1	system error (see errno.h)
2	file not found (check environment variables)
3-99	system error (see errno.h)
100	duplicate record
101	file not open
102	illegal argument
103	illegal key description
104	too many files open
105	bad isam file format
106	exclusive access required
107	record locked
108	key already exists
109	is primary key
110	end/begin of file
111	no record found
112	no current record
113	file locked
114	file name too long
115	cannot create lock file
116	cannot allocate memory
117	bad custom collating
201	NULL file pointer (pdbhead)
203	file not open

INDEX

INDEX

INFOFLEX®, INC

I N F O F L E X - 4 G L

ToolFlex Function Guide

Printed in U.S.A. on May 2000

Name

bflush - flush internal I/O buffer to the terminal screen

Syntax

bflush( )

Description

Bflush flushes to the terminal screen the internal I/O buffer filled by one of these non-flushing output routines:

boxline
boxrev
bshow
bshowxy
clrbox
clreol
clreos
clrpage
clrscr
gotoxy
putkey

Example

```
gotoxy( 0,3);
bshow( " Writing many lines ");
bflush();
```

Return Value

None.

Name

box - display a box on the screen

Syntax

boxline(urow, lcol, brow, rcol)
int urow, lcol, brow, rcol;

boxrev(urow, lcol, brow, rcol)
int urow, lcol, brow, rcol;

Description

Boxline displays a box using a single line border.

Boxrev displays a box with a reverse attribute background.

Urow, *lcol*, *brow*, and *rcol* define the upper row, left column, bottom row, and right column, respectively, of the border of the box.

Return Value

None.

Name

buftostr - convert a value to a string

Syntax

buftostr(s1, s2, type, length, dec, fmt)
char *s1;
char *s2;
int type;
int length;
int dec;
int fmt;

strtobuf(s1, s2, type, length, fmt)
char *s1;
char *s2;
int type;
int length;
int fmt;

Description

Buftostr converts the value pointed to by *s1* to the character string *s2*.

Strtobuf converts the character string *s1* to a value pointed to by *s2*.

Type is the data type of the value and is defined in **fxtypes.h**:

CHARTYPE	character string
SHORTTYPE	short integer
INTTYPE	integer
LONGTYPE	long integer
DOUBLETYPE	double-precision floating point number
DECTYPE	decimal structure
MONEYTYPE	money
SERIALTYPE	serial long integer
DATETYPE	date
TIMETYPE	AM/PM style time
MTIMETYPE	military time

Length is the length of the string buffer (*s2* for **buftostr** or *s1* for **strtobuf**). *length* must be at least 1 byte less than the size of the buffer for null termination. If necessary, *s2* of *type* **CHARTYPE** will be truncated on the right to fit within *length*.

In **buftostr** *dec* is the number of digits to the right of the decimal point for **DOUBLETYPE**, **FLOATTYPE**, or **DECTYPE**. If *dec* is greater than zero, *s2* will be truncated on the left, if necessary, to fit the result within *length*.

Fmt is a format flag. The format flags that are meaningful to **buftostr** are defined in **fxcon.h**:

F_BLANKFILL	Blank fill *s2* if *s1* is 0.
F_RIGHT	Right justify *s2* within *length*. This also works for **strtobuf**.
F_LEFT	Left justify *s2* within *length*.
F_COMMA	Comma separate groups of three digits left of a decimal point. This macro can be added in with one of the justification macros.

In **strtobuf** if *fmt* has the value of **F_PHONE** and *type* is **CHARTYPE**, then *s1* is checked for valid telephone format.

Return Value

- -1 Conversion failed (**strtobuf** only).
- 0 Conversion successful.

Name

chkent - check if a field is data enterable

Syntax

int chkent(pscrfield)
SCRFIELD *pscrfield;

Description

Chkent checks whether the screen field pointed to by *pscrfield* is data enterable according to the current mode. For example, if the mode is **CHANGE** (**flexmode** = **CHANGEMODE**) and the field has an attribute of **noupdate**, **chkent** will return 1. If the mode is **ADD** (**flexmode** = **ADDMODE**) and the attribute **noentry** is in effect for the field, a 1 will be returned.

Return Value

1 Field is not enterable.
0 Field is enterable.

Name

clr - clear parts of the screen display

Syntax

clrscr( )

clrpage(pscrhead)
SCRHEAD *pscrhead;

clreos(row, col)
int row, col;

clreol( )

clrrng(row, nlines)
int row, nlines;

clrbox(urow, lcol, brow, rcol)
int urow, lcol, brow, rcol;

Description

Clrscr clears the entire screen display.

Clrpage clears the screen region occupied by a **SCREENFLEX** screen image.

Clreos clears from the current cursor position, specified by *row* and *col*, to the end of the screen display.

Clreol clears from the current cursor position to the end of the line.

Clrrng clears a range of screen rows starting at *row* and clearing *nlines* number of lines.

Clrbox clears a box region on the CRT. The area of the box is defined by *urow*, the upper row, *lcol*, the leftmost column , *brow*, the bottom row, and *rcol*, the rightmost column. The region is cleared by overwriting blanks.

Example

To clear line *n*, use the following:

```
gotoxy( n, 0);
clreol();
```

or

```
clrrng( n, 1);
```

Return Value

None.

Name

compare - compare data from one field to another

Syntax

compare(pfield1, pfield2)
FIELD *pfield1, *pfield2;

Description

Compare compares the contents of *pfield1* and *pfield2*. This function will convert data types if necessary. Also, one of the arguments may be a C language variable provided it is of the same type and length. The source and the destination can be either a screen field, a database field, or a C variable. However, you cannot use it to compare a C variable to a C variable.

Examples

There are several ways to compare values into a screen or table field via user functions. A summary of the various ways are shown below:

```
if ($table.fieldname == 14)
```

or

```
cnum = 14;
if (compare( &cnum, @table.fieldname) == 0)
```

The first is more efficient when dealing with numerics. However, when dealing with character strings compare is the proper method to use.

```
if ( compare( "teststring", @table.fieldname) == 0)
```

Note that money fields are stored in units of cents.

Limitation

Comparisons to a screen **array** variable can only be done when the screen is active (currently displayed).

Return Value

= 0 *pfield1* is equal to *pfield2*.
> *pfield1* is greater than *pfield2*.
< *pfield1* is less than *pfield2*.

See Also

move, buftostr

Name

datestr - convert date internal format to date display format

Syntax

datestr(numdate, sdate, format)
long numdate;
char *sdate;
char *format;

rdatestr(numdate, sdate, type)
long numdate;
char *sdate;
int type;

strdate(sdate, numdate, format)
char *sdate;
long *numdate;
char *format;

rstrdate(sdate, numdate, type)
char *sdate;
long *numdate;
int type;

Description

Datestr converts a date from internal format, *numdate*, to display format, *sdate*. Internal format is the number of days since December 31, 1899. The *format* argument determines the format for *sdate*. If *format* is null, then the environment variable **FXDATE** determines the format. If **FXDATE** is undefined, then **MM/DD/YY** is used. Other possible formats are described below.

Rdatestr also converts a date from internal format, *numdate*, to display format, *sdate*. The *type* argument determines the format of *sdate* and may be **DATETYPE**, **EDATETYPE**, or **YDATETYPE**. **DATETYPE** formats to **MM/DD/YY**, **EDATETYPE** formats to **DD/MM/YY**, and **YDATETYPE** formats to **YY/MM/DD**.

Strdate converts a date from display format, *sdate*, to internal format, *numdate*. The *format* argument specifies the format of *sdate*. *Format* may be **MM/DD/YY**, **DD/MM/YY**, or **YY/MM/DD**. If *format* is null, the format defaults in the same manner as for **datestr**. A blank or zero length *sdate* will null the internal date.

Rstrdate also converts a date from display format, *sdate*, to internal format, *numdate*. The *type* argument specifies the format of *sdate* and may be **DATETYPE**, **EDATETYPE**, or **YDATETYPE**. **DATETYPE** formats to **MM/DD/YY**, **EDATETYPE** formats to **DD/MM/YY**, and **YDATETYPE** formats to **YY/MM/DD**. A blank or zero length *sdate* will null the internal date.

FORMAT Options

The *format* argument for **datestr()** has a number of options. For example, you may use the format **Mmm dd, yyyy Ddd** to display the date as "Jan 23, 1992 Tue". These formating options are described below.

Format	Description
mm	displays a two digit representation of the month
mmm	displays a three letter abbreviation of the month
dd	displays a two digit representation of the day
ddd	displays a three letter abbreviation of the day
yy	displays a two digit representation of the year
yyyy	displays a four digit representation of the year

Return Value

R_NULL	Internal input date is null. (For **datestr** or **rdatestr** only. Macro definition in **fxcon.h**.)
R_ZERO	Internal input date is less than or equal to 0. (For **datestr** or **rdatestr** only. Macro definition in **fxcon.h**.)
-1	Illegal *format*, *type*, or *sdate*.
0	Conversion successful.

Name

dclr - clear one or more fields of the table buffer

Syntax

dclrrec(pdbhead)
DBHEAD *pdbhead;

dclrfld(pdbfield)
DBFIELD *pdbfield;

dclrrng(pdbfield1, pdbfield2)
DBFIELD *pdbfield1;
DBFIELD *pdbfield2;

Description

Dclrrec clears the table buffer pointed to by *pdbhead*.

Dclrfld clears the field in the table buffer pointed to by *pdbfield*.

Dclrrng clears a range of fields starting at *pdbfield1* and ending at *pdbfield2*.

Return Value

None.

See Also

sclr, tclr

Name

dmap - copy one or more fields from the screen buffer to the table buffer

Syntax

dmaprec(pdbhead, pscrhead)
DBHEAD *pdbhead;
SCRHEAD *pscrhead;

dmapfld(pdbhead, pscrfield)
DBHEAD *pdbhead;
SCRFIELD *pscrfield;

dmaprng(pdbhead, pscrfield1, pscrfield2)
DBHEAD *pdbhead;
SCRFIELD *pscrfield1, *pscrfield2;

Description

Dmaprec copies all fields from the screen buffer pointed to by *pscrhead* to the table buffer *pdbhead*.

Dmapfld copies the data of the screen buffer field pointed to by *pscrfield* to the table buffer pointed to by *pdbhead*.

Dmaprng copies a range of fields starting with *pscrfield1* and ending with *pscrfield2* of the screen buffer to the table buffer *pdbhead*.

Return Value

None.

See Also

smap, tmap

Name

error - error codes

Description

When a TOOLFLEX file management function fails (functions which start with a **fm**), it sets the global variable **iserrno** to one of these codes defined in **fxisam.h**:

1	system error (see errno.h)
2	file not found (check environment variables)
3-99	system error (see errno.h)
100	duplicate record
101	file not open
102	illegal argument
103	illegal key description
104	too many files open
105	bad isam file format
106	exclusive access required
107	record locked
108	key already exists
109	is primary key
110	end/begin of file
111	no record found
112	no current record
113	file locked
114	file name too long
115	cannot create lock file
116	cannot allocate memory
117	bad custom collating
201	NULL file pointer (pdbhead)
203	file not open

Name

flex - load and run a flex program

Syntax

flex(argc, argv)
int argc;
char *argv[];

flexcmd(command)
char *command;

flexload(picname)
char *picname;

Description

Flex initializes the terminal, then loads a binary screen display or report definition **.pic** file and runs its **SCREEN** or **REPORT** section. **Flex** takes **main**-like parameters, *argc* and *argv*, that would be the same as the arguments to the **flex** command. **Flex** may be called recursively to zoom or pop-up other screens from the calling screen. The recursively called screen must be defined in the same **.flx** source file as the calling screen.

Flexcmd performs the same function as **flex** except the arguments are specified as a *command* string.

Flexload performs the same function as **flex** except the **SCREEN** or **REPORT** section is not run. This function is typically used for Infoflex programs which only have a **TABLES** and **INSTRUCTIONS** section. The **TABLES** section is compiled into a **.pic** file called *picname*, which is the argument to **flexload**. The developer will supply the **main** function and must call **flexload** before using any table management or screen management functions.

Limitations

When recursively calling screens, the called screen must reside in the same source file as the calling screen. Also the first argument must be **flex**. **Flex** functions may be called from outside userexits or from within function key userexits.

Return Value

Last flexkey pressed upon exiting flex() or flexcmd(). There is no return value for flexload().

See Also

fxinit

Example

```
flexcmd( "flex bkinput -f tbvenpop cdsqa N");
```

will load the Infoflex program **bkinput** and then run the screen **tbvenpop** with CHANGE, DELETE, SEARCH, and ADD MODES enabled.

Name

fmaddindex - add an index to a table

Syntax

fmaddindex(pdbhead, pdbindex)
DBHEAD *pdbhead;
DBINDEX *pdbindex;

Description

Fmaddindex adds the index pointed to by *pdbindex* to the table associated with the table buffer *pdbhead*.

Return Value

- -2 Invalid *pdbindex* structure.
- -1 Failed to open table, if table was not already open, or failed to create index. The global variable **iserrno** will be set with one of the values listed in **error**(**T**).
- 0 Index created successfully.

See Also

fmdelindex

Name

fmbegin - defines the beginning of a transaction

Syntax

fmbegin()

Description

Fmbegin must be called before the first file management function call (*fmxxxx*) within a transaction.

Limitations

This function is currently only available for C-ISAM users (call for current status).

Return Value

- 1 Failed condition; **iserrno** is set (see **error (T)**).
0 Successful condition.

See Also

fmbegin, fmcommit, fmrollback, fmlogopen, fmlogclose, fmrecover

Name

fmbldall - build a table and all its indices

Syntax

fmbldall(pdbhead)
DBHEAD *pdbhead;

Description

Fmbldall will build the table specified by *pdbhead*, creating all indices defined within the table buffer structure. Where the table already exists, it is erased prior to building. This routine is effective in deleting the contents of existing files.

Return Value

- -2 **Fmaddindex** failed.
- -1 Null *pdbhead* or **fmbuild** failed.
- 0 Successful.

See Also

fmbuild, fmaddindex, fmerase

Name

fmbuild - create a table

Syntax

fmbuild(pdbhead, pdbindex, mode)
DBHEAD *pdbhead;
DBINDEX *pdbindex;
int mode;

Description

Fmbuild will create the table specified by *pdbhead* and create the table's primary index as specified by *pdbindex*.

The created table will be left open in the specified *mode*. See **mode**(**T**) for the description of the table access modes.

After **fmbuild**, you may create secondary indices with the **fmaddindex** function.

Return Value

-2 Invalid *pdbindex* structure.
-1 Table creation failed. The global variable **iserrno** will be set with one of the values listed in **error**(**T**).
>=0 File descriptor of open created table.

See Also

fmbldall, fmaddindex, fmerase

Name

fmclose - close a table

Syntax

fmclose(pdbhead)
DBHEAD *pdbhead;

Description

Fmclose closes the table specified by the table buffer *pdbhead*.

Return Value

- 1 Null *pdbhead* or close failed. In the latter case **iserrno** is set (see **error**(**T**)).
0 Close successful or table already closed.

See Also

fmopen

Name

fmcommit - causes all File Management functions since the last call to **fmbegin** to take effect.

Syntax

fmcommit()

Description

Fmcommit will cause all changes to INFOFLEX files within the transaction to occur. The transactions begins upon calling the function **fmbegin()**. All locks held for transaction are released upon completion of the committing process.

Limitations

This function is currently only available for C-ISAM users (call for current status).

Return Value

- -1 Failed condition; **iserrno** is set (see **error (T)**).
- 0 Successful condition.

See Also

fmbegin, fmcommit, fmrollback, fmlogopen, fmlogclose, fmrecover

Name

fmcur - test if there is a current record for a table

Syntax

fmcurchk(pdbhead)
DBHEAD *pdbhead;

fmcurclr(pdbhead)
DBHEAD *pdbhead;

Description

Fmcurchk determines if the table specified by *pdbhead* has been opened and is currently positioned on a record. A record is positioned via the **fmstart**, **fmfind**, or **fmread** functions.

Fmcurclr clears the current position flag so the next time fmsave() is called a new record will be written.

Return Value

0 No current record exists.
1 Current record exists.

See Also

fmstart, fmfind, fmread

Name

fmdelcurr - delete the current record of a table

Syntax

fmdelcurr(pdbhead)
DBHEAD *pdbhead;

Description

Fmdelcurr deletes the current record associated with the table buffer pointed to by *pdbhead*. The current record is the last record of the table accessed with **fmfind** or **fmread**. After **fmdelcurr** the current record will be the record *previous* to the one deleted.

Return Value

- 1 Delete failed; **iserrno** is set (see **error**(**T**)).
0 Delete successful.

See Also

fmdelete, fmfind, fmread

Name

fmdelete - delete the record with a given primary index value

Syntax

fmdelete(pdbhead)
DBHEAD *pdbhead;

Description

Fmdelete deletes the record with the values for its primary index defined in the table buffer *pdbhead*. The primary index must be an index that does not allow duplicate entries. You may not use this function with files created with SQLFLEX or INFORMIX-SQL.

Return Value

- -1 Delete failed; **iserrno** is set (see **error**(**T**)).
- 0 Delete successful.

See Also

fmdelcurr

Name

fmdelindex - remove an index

Syntax

fmdelindex(pdbhead, pdbindex)
DBHEAD *pdbhead;
DBINDEX *pdbindex;

Description

Fmdelindex deletes the index specified by *pdbindex* from the table specified by *pdbhead*. The table may be open but need not be.

Return Value

- -2 Invalid *pdbindex* structure.
- -1 Failed to open table, if table was not already open, or failed to delete index. The global variable **iserrno** will be set with one of the values listed in **error**(**T**).
- 0 Index deleted successfully.

See Also

fmaddindex

Name

fmdelrec - delete a record from the table identified by its record number

Syntax

fmdelrec(pdbhead, recd)
DBHEAD *pdbhead;
long recd;

Description

Fmdelrec will delete record number *recd* in table *pdbhead*. If that record happens to be the "current" record, the current record will not change.

Return Value

- 1 Delete failed; **iserrno** is set (see **error**(**T**)).
0 Delete successful.

See Also

fmdelcurr, fmdelete, fmfind, fmread

Name

fmdictinfo - get table parameters

Syntax

fmdictinfo(pdbhead, pdictinfo)
DBHEAD *pdbhead;
struct dictinfo *pdictinfo;

Description

Fmdictinfo returns table parameters contained in the *pdictinfo* structure which is defined in **fxisam.h**:

```
struct dictinfo
    {
    short di_nkeys;      /* number of indices defined */
    short di_recsize;    /* data record size */
    short di_idxsize;    /* index record size */
    long di_nrecords;    /* number of records in table */
    };
```

The table is specified by *pdbhead*.

Return Value

- 1 Access failed; **iserrno** is set (see **error**(**T**)).
0 Access successful.

See Also

fmindexinfo

Name

fmerase - remove a table

Syntax

fmerase(pdbhead)
DBHEAD *pdbhead;

Description

Fmerase will remove the table specified by *pdbhead* from its database.

Return Value

- 1 Could not close the table, if open, or could not remove it. In the former case **iserrno** is set (see **error**(**T**)).
0 Removal successful.

See Also

fmbuild

Name

fmerrmsg - display a message based on the last table management function call

Syntax

fmerrmsg(pdbhead)
DBHEAD *pdbhead;

Description

Fmerrmsg displays a message based on the last table management function call accessing the table specified by *pdbhead*. **Fmerrmsg** places the message at line 23 of the screen, rings the bell, and waits for a carriage return to continue.

Return Value

None.

See Also

message

Name

fmfind - find a record based on the given mode and index

Syntax

fmfind(pdbhead, pdbindex, mode)
DBHEAD *pdbhead;
DBINDEX *pdbindex;
int mode;

Description

Fmfind will do a combination of both the functions **fmstart** and **fmread**. If the *pdbindex* is being used for the first time, the **fmfind** function will perform the **fmstart** function. After the **fmstart** is done the **fmread** will be performed. Th acceptable **fmfind** *modes* are the same as for the **fmread**.

Return Value

- -1 Find failed; **iserrno** is set (see **error**(**T**)).
- 0 Find successful.

See Also

fmstart, fmread

Name

fmflush - flushes data and indexes to disk

Syntax

fmflush(pdbhead)
DBHEAD *pdbhead;

Description

Fmflush ensures that all data and indexes associated with the table *pdbhead* are written to disk.

Return Value

- 1 Flush failed; **iserrno** is set (see **error**(**T**)).
0 Flush successful.

See Also

fmclose

Name

fmindexinfo - get information about a table's indices

Syntax

fmindexinfo(pdbhead, pdbindex, pkeydesc)
DBHEAD *pdbhead;
DBINDEX *pdbindex;
struct keydesc *pkeydesc;

Description

Fmindexinfo returns parameters associated with the index pointed to by *pdbindex* of the table specified by *pdbhead*. The index parameters are loaded into the *pkeydesc* structure defined in **fxisam.h**:

```
struct keydesc
    {
    short k_flags; /* flags */
    short k_nparts;/* number of fields in index */
                   /* index fields */
    struct keypart k_part[NPARTS];
    short k_len;  /* length of whole index(internal use)*/
    long k_rootnode;/* pointer to rootnode(internal use)*/
    };

struct keypart
    {
    short kp_start;/* starting byte of index part */
    short kp_leng; /* length in bytes */
    short kp_type; /* type of index part */
    };
```

The **k_flags** element specifies the type of index (these macros are defined in **fxisam.h**):

ISNODUPS	no duplicates allowed.
ISDUPS	duplicates allowed.
DCOMPRESS	compresses duplicates.
LCOMPRESS	compresses leading characters.
TCOMPRESS	compresses trailing characters.
COMPRESS	all forms of compression.

Return Value

-1	Invalid *pdbindex* structure.
0	Successful.

See Also

fmdictinfo

Name

fmload - dynamically load table information into memory
fmunload - unload dynamically loaded table information

Syntax

fmload(tablelist)
char *tablelist;

fmunload()

Description

Fmload loads the table information dynamically rather than through a **.pic** file. This is an alternate method to using an Infoflex source file with a **TABLES** section.

When using **fmload**, your program will look like a normal C program without any Infoflex section names, @variables, or $variables. You will also need to supply your own **main** function.

Fmload must be called before any other file management routine (**fm***xxxx*) and should only be called once in a program.

Upon an error, the program will abort with an appropriate error message.

Example

```
fmload( "vendor customer accounts");
```

will load the dictionary information for the three tables **vendor**, **customer**, and **accounts**.

The following example does the same as above except the customer file is loaded twice under an *alias* name **cust2**.

```
fmload( "vendor customer accounts alias cust2");
```

This technique is used so the same table can be opened more than once in order to access it in different ways at the same time.

To load the entire dictionary, use the keyword "alltables" as the *tablelist* argument.

Limitations

This function may NOT be used in flex programs (*.flx).

Return Value

None.

Name

fmlock - creates a lock on the entire table.

Syntax

fmlock(pdbhead)
DBHEAD *pdbhead;

Description

Fmlock places a read only lock on the entire table *pdbhead*. It works only for tables opened for manual locking (ISMANULOCK). The table may be unlock by calling the **fmunlock** function.

Return Value

- 1 Lock failed; **iserrno** is set (see **error**(**T**)).
0 Lock successful.

See Also

fmunlock, fmrelease

Name

fmlogclose - closes the transaction log file.

Syntax

fmlogclose()

Description

Fmlogclose closes the log file opened by **fmlogopen()**.

Limitations

This function is currently only available for C-ISAM users (call for current status).

Return Value

- 1 Failed condition; **iserrno** is set (see **error (T)**).
0 Successful condition.

See Also

fmbegin, fmcommit, fmrollback, fmlogopen, fmlogclose, fmrecover

Name

fmlogopen - Opens th etransaction log file for all subsequent File Management calls to record changes.

Syntax

fmlogopen(logname)
char *logname;

Description

Fmlogopen must specify a *logname* where transactions are to be stored. This log file must already exist prior to calling this function.

Limitations

This function is currently only available for C-ISAM users (call for current status).

Return Value

- -1 Failed condition; **iserrno** is set (see **error (T)**).
- 0 Successful condition.

See Also

fmbegin, fmcommit, fmrollback, fmlogopen, fmlogclose, fmrecover

Name

fmopen - open a table

Syntax

fmopen(pdbhead, mode)
DBHEAD *pdbhead;
int mode;

Description

Fmopen opens the table specified by *pdbhead* in the specified *mode*. See **mode**(**T**) for the description of the table access modes.

Return Value

`< 0` Open failed.
`>=0` File descriptor of the opened table.

See Also

fmclose

Name

fmread - read a record into the table buffer

Syntax

fmread(pdbhead, mode)
DBHEAD *pdbhead;
int mode;

Description

Fmread reads a record from the table specified by *pdbhead*. The record is read sequentially or randomly as indicated by one of these values for *mode* defined in **fxisam.h**:

ISCURR	reads the current record.
ISFIRST	reads the first record.
ISLAST	reads the last record.
ISNEXT	reads the next record.
ISPREV	reads the previous record.
ISEQUAL	reads the record equal to the search value.
ISGREAT	reads the first record greater than the search value.
ISGTEQ	reads the first record greater than or equal to the search value.

The search value will be those values in the fields of *pdbhead* that comprise the index specified by the most recent **fmstart** or **fmfind**.

To Lock the record read by **fmread**, add ISLOCK to one of the retrieval modes (for example ISEQUAL+ISLOCK). In order to use this locking feature the file must have been opened using the ISMANULOCK locking mode (see **mode**(T). The record remains locked until you unlock it with **fmrelease**.

You may use the **fmread** to read specific records via the record number. To do this you must first call **fmstart** with a NULL argument for *pdbindex*. Then assign the global variable **isrecnum** with the record number you wish to locate and call **fmread** with an ISEQUAL mode.

Example

```
fmread( pdbhead, ISFIRST+ISLOCK);
```

Return Value

-1	Read failed; **iserrno** is set (see **error**(**T**)).
0	Read successful.

See Also

fmstart, fmfind

Name

fmrecover - re-updates INFOFLEX files with transactions from the log file.

Syntax

fmrecover()

Description

Fmrecover is used to update a backup copy of your INFOFLEX files with a log file generated since the time of the backup. The log file must already be opened by a call to **fmlogopen**. Also, the function should finish executing before anyone is allowed to access the files.

Limitations

This function is currently only available for C-ISAM users (call for current status).

Return Value

- -1 Failed condition; **iserrno** is set (see **error (T)**).
- 0 Successful condition.

See Also

fmbegin, fmcommit, fmrollback, fmlogopen, fmlogclose, fmrecover

Name

fmrelease - unlock the records of a table

Syntax

fmrelease(pdbhead)
DBHEAD *pdbhead;

Description

Fmrelease removes the locks placed on any records locked by the **fmfind** or **fmread** functions. The records belong to the table specified by *pdbhead*.

A table must be opened with the **ISMANULOCK** mode to make use of the **fmrelease** function.

Return Value

- -1 Release failed; **iserrno** is set (see **error**(**T**)).
- 0 Release successful.

See Also

fmopen, fmfind, fmread

Name

fmrename - rename a table

Syntax

fmrename(oldname, newname)
char *oldname;
char *newname;

Description

Fmrename changes the name of a table, *oldname*, to the new name, *newname*. These are the names of the data files without the **.dat** or **.idx** extensions. These names are not necessarily the dictionary table names.

Return Value

- 1 Rename failed; **iserrno** is set (see **error**(**T**)).
0 Rename successful.

Name

fmrewcurr - rewrite the current record of a table

Syntax

fmrewcurr(pdbhead)
DBHEAD *pdbhead;

Description

Fmrewcurr updates the current record of the table specified by the table buffer *pdbhead* and with the data in the table buffer. The current record is the last record of the table accessed with **fmfind** or **fmread**.

Return Value

- -1 Rewrite failed; **iserrno** is set (see **error**(**T**)).
- 0 Rewrite successful.

See Also

fmrewrite, fmfind, fmread

Name

fmrewrec - rewrite a record from the table identified by its record number

Syntax

fmrewrec(pdbhead, recd)
DBHEAD *pdbhead;
long recd;

Description

Fmrewrec will rewrite record number *recd* in table *pdbhead*. If that record happens to be the "current" record, the current record will not change.

Return Value

- 1 Delete failed; **iserrno** is set (see **error**(**T**)).
0 Delete successful.

See Also

fmdelcurr, fmdelete, fmfind, fmread

Name

fmrewrite - rewrite the record with a given primary index value

Syntax

fmrewrite(pdbhead)
DBHEAD *pdbhead;

Description

Fmrewrite updates a record of the table specified by *pdbhead* and with the data in the table buffer. The record is located by the values in the table buffer for its primary index. The primary index must be an index that does not allow duplicate entries. You may not use this function on files created using SQLFLEX or INFORMIX-SQL.

Return Value

- 1 Rewrite failed; **iserrno** is set (see **error**(**T**)).
0 Rewrite successful.

See Also

fmrewcurr

Name

fmrollback - Cancels all File Management calls made since the last call to **fmbegin()**.

Syntax

fmrollback()

Description

Fmrollback returns any records modified since the last call to **fmbegin()** to their original unmodified state. To use the **fmrollback()** function you must include the ISTRANS parameter as part of the mode in the **fmopen** call. Also you may not roll back the following calls: **fmbuild, fmaddindex, fmdelindex, fmsetunique, fmuniqueid, fmrename, or fmerase**.

Limitations

This function is currently only available for C-ISAM users (call for current status).

Return Value

- -1 Failed condition; **iserrno** is set (see **error (T)**).
- 0 Successful condition.

See Also

fmbegin, fmcommit, fmrollback, fmlogopen, fmlogclose, fmrecover

Name

fmsave - save a table record in the appropriate way

Syntax

fmsave(pdbhead)
DBHEAD *pdbhead;

Description

Fmsave is an optimized record write routine. **Fmwrite** is called if the last **fmfind** call was unsuccesful. **Fmrewcurr** is called if the last **fmfind** call was successful.

Return Value

- -1 Save failed; **iserrno** is set (see **error**(**T**)).
- 0 Save successful.

See Also

fmwrite, fmrewcurr, fmfind

Name

fmsetserial - set the serial number in the table buffer

Syntax

fmsetserial(pdbhead)
DBHEAD *pdbhead;

Description

Fmsetserial sets the serial number field to a unique number. A unique number is only assigned if the current value of the serial field is 0. The unique number is one greater than the last one generated by calling **fmsetserial**.

Return Value

None.

Name

fmsetunique - set the value of the unique identifier for a table

Syntax

fmsetunique(pdbhead, recnum)
DBHEAD *pdbhead;
long recnum;

Description

Fmsetunique sets a new starting value for the unique number field of the next record created for the table specified by *pdbhead*. This starting value will be *recnum*. If *recnum* is less than the current value of the unique number field, then **fmsetunique** has no effect.

Return Value

None.

See Also

fmuniqueid

Name

fmstart - select the index for subsequent operations

Syntax

fmstart(pdbhead, pdbindex, keylength, mode)
DBHEAD *pdbhead;
DBINDEX *pdbindex;
int keylength;
int mode;

Description

Fmstart specifies and finds the index (*pdbindex*) for subsequent **fmread** calls. The **fmstart** does not, however, read the record into the table buffer. If a record is found by the **fmstart** function, a call to **fmread** with a mode of ISCURR is required to read the records data from the disk into the table's buffer.

If you want to locate a record by the entire length of the key then set the *length* argument either to zero or the entire length of the key. To locate a record based on part of the key, set *Length* to the number of initial bytes of the index you want to use.

Mode defines the relation to the search value of the first record to be read with *fmread*. *Mode* can have one of these values defined in **fxisam.h**:

ISFIRST	finds the first record. Search value is irrelevant.
ISLAST	finds the last record. Search value is irrelevant.
ISEQUAL	finds the record equal to the search value.
ISGREAT	finds the first record greater than the search value.
ISGTEQ	finds the first record greater than or equal to the search value.

The search values for the index fields must be assigned to the appropriate fields of *pdbhead* prior to the call.

If the mode ISFIRST or ISLAST are used, **fmstart** ignores the values in the table buffer and the *Length* argument.

If *pdbindex* is NULL, the records will be read sequentially. While in sequential mode, records may be read by record number setting the global variable **isrecnum** to the desired record number and calling **fmread** with an ISEQUAL mode.

Note that you only need to use the **fmstart** function when you want to change an index or use part of a key as the search criteria. You do not need to use the **fmstart** function before each **fmread** call.

Return Value

- -1 Index selection failed or could not position for read based on search value. In the latter case the index selection can be successful. Upon an error **iserrno** is set (see **error**(**T**)).
- 0 Both index selection and positioning for read are successful.

See Also

fmread, fmfind

Name

fmstructview - map a set of database fields to a structure

Syntax

fmstructview(pdbhead, pdbview, vwnum, vwstruct)
DBHEAD *pdbhead;
struct dbview *pdbview;
int vwnum;
char *vwstruct;

Description

Fmstructview allows you to specify a C structure where a database table will be read to and written from. *Pdbhead* argument specifies the table and *vwstruct* the C structure where the data will be stored. The programmer may then use *vwstruct* to access the table's fields. If the *pdbview* argument is not used (or NULL) then the structure *vwstruct* must have every table field declared and in exactly the same order. The *pdbview* argument allows the programmer to specify the format of the structure. The fields selected for the view are listed in the **dbview** array pointed to by *pdbview*. This is the **dbview** structure is assigned:

```
struct dbview empview[] =
     {
     {"code"},
     {"lname"},
     {"fname"},
     {"hire_date"},
     {"socno"}
     };
```

Dbview is assigned the SQL field names from a table. *Vwstruct* must match this list.

Vwnum argument is only used if there *pdbview* is specified and is the number of fields in *pdbview*.

For null termination, the character string elements of *vwstruct* must be one byte longer than the corresponding table field sizes.

Limitations

Once you have defined a view with **fmstructview**, you must access fields via *vwstruct* and not *pdbhead*. Any assignments to fields of *pdbhead* will be overwritten by *vwstruct* with a subsequent table buffer output file management call.

Also, for portability you should not use **short** data types within your structures.

Return Value

0 successful.
-1 invalid field name in the view.

Example

See Reference Manual Appendix: Sample ISAMFLEX Program

Name

fmuniqueid - get the next unique number for a table

Syntax

fmuniqueid(pdbhead, recnum)
DBHEAD *pdbhead;
long *recnum;

Description

Fmuniqueid will generate a unique number for the table pointed to by *pdbhead*. *Recnum* will point to this unique value.

Return Value

- 1 Access failed; **iserrno** is set (see **error**(**T**)).
0 Access successful.

See Also

fmwrite, fmsetunique

Name

fmunlock - removes a lock on a table.

Syntax

fmunlock(pdbhead)
DBHEAD *pdbhead;

Description

Fmunlock removes the table lock set by **fmlock**.

Return Value

- 1 Unlock failed; **iserrno** is set (see **error**(**T**)).
0 Unlock successful.

See Also

fmlock, fmrelease

Name

fmwrcurr - write a table record making it the current record

Syntax

fmwrcurr(pdbhead)
DBHEAD *pdbhead;

Description

Fmwrcurr writes the record data contained in the *pdbhead* structure to the table specified by *pdbhead*. The record written becomes the current record. The record's unique number field is automatically incremented if the field value is 0 going in. You should 0 a record with **dclrrec**(**T**) before filling in the fields of *pdbhead*.

Return Value

- 1 Write failed; **iserrno** is set (see **error**(**T**)).
0 Write successful.

See Also

fmwrite

Name

fmwrite - write a new record into a table

Syntax

fmwrite(pdbhead)
DBHEAD *pdbhead;

Description

Fmwrite writes the field data in the *pdbhead* structure to a new record of the table specified by *pdbhead*. The record's unique number field is automatically incremented if the field value is 0 going in. You should 0 a record with **dclrrec**(**T**) before filling in the fields of *pdbhead*.

This function changes the global variable **isrecnum** to the record number of the recently written record.

Return Value

- -1 Write failed; **iserrno** is set (see **error**(**T**)).
- 0 Write successful.

See Also

fmwrcurr, dclrrec

Name

fxabort - exit an Infoflex program

Syntax

fxabort(status)
int status;

Description

Fxabort will take the necessary actions to terminate a Infoflex program. Passing a *status* of 0 will result in a normal abort; passing -1 will result in an abnormal abort. When a *status* of -1 is passed, the program is aborted passing 1 back to the calling program or operating system.

Limitations

Fxabort should not be called unless the terminal has been initialized using functions **fxinit**, **flex**, **flexcmd**, or **flexload**.

Return Value

None.

Name

fxaccept - prompt for a screen field

Syntax

fxaccept(pscrfield)
SCRFIELD *pscrfield;

Description

Fxaccept prompts for a screen field based on parameters in the **SCRFIELD** and **SCRTRANS** structures. These structures are generated from your **.flx** source field.

Return Value

- -1 Pscrfield is NULL.
- 0 Fxaccept successful.
- 1 The field is noentry or noupdate.

See Also

prompt

Name

fxasave - default function for saving an array screen line to a table

Syntax

fxasave( )

Description

Fxasave will modify or add table records associated with the current screen array line. This function will do the following:

- Check for required fields
- Execute the abeforesave() userexit
- Save the record
- Execute the aaftersave() userexit

Limitations

Fxssave may NOT be called from a **abeforesave** userexit or **aaftersave** userexit.

Return Value

- -1 Save failed. If the error is a table access error **iserrno** is set (see **error**(**T**)).
- 0 Save successful.

See Also

fxssave

Name

fxinit - initialize the terminal

Syntax

fxinit( )

Description

Fxinit initializes the terminal using the **TERMFLEX** (Infoflex's equivalent of **termcap**) definitions. **Fxinit** must be called prior to using any screen I/O functions such as **show**, **gotoxy**, **clrscr**, etc.

Return Value

None.

Name

fxround - round a value to the specified number of decimal places

Syntax

double fxround(value, places)
double value;
unsigned places;

Description

Fxround rounds a double-precision float *value* to a specified number of decimal *places* right of the decimal point.

Return Value

None.

Name

fxspawn - call another external program using variable arguments and return to the calling program

Syntax

fxspawn(argc, argv)
int argc;
char *arg[];

Description

Fxspawn calls an external program using **main**-like *argc* and *argv*. When the external program terminates, program control will return to the code following the **fxspawn** call.

Return Value

None.

See Also

fxvexec, fxsystem

Name

fxssave - default function for saving a non-array screen data to a table

Syntax

fxssave( )

Description

Fxssave will modify or add table records associated with a screen header. This function will do the following:

- Check for required fields
- Execute the beforesave() userexit
- Save the record
- Execute the aftersave() userexit

Limitations

Fxssave may NOT be called from a **beforesave** userexit or **aftersave** userexit.

Return Value

- -1 Save failed. If the error is a table access error **iserrno** is set (see **error**(**T**)).
- 0 Save successful.

See Also

fxasave

Name

fxsystem - do a system call from an Infoflex program

Syntax

fxsystem(command)
char *command;

Description

Fxsystem executes *command* as an external program. When the external program terminates, program control will return to the code following the **fxsystem** call.

Return Value

None.

See Also

fxvexec

Name

fxvexec - calls an external program but does not return

Syntax

fxvexec(argc, argv)
int argc;
char *arg[];

Description

Fxvexec calls an external program using **main**-like *argc* and *argv*. This is a program overlay, and in effect, the calling program terminates with the **fxvexec**.

Return Value

None.

See Also

fxspawn, fxsystem

Name

getdf - get the field value from the table buffer

putdf - put a field value in the table buffer

getsf - get the field value from the screen buffer

putsf - put a value into a field of the screen buffer

Syntax

getdf(pvalue, pdbfield)
char *pvalue;
DBFIELD *pdbfield;

putdf(pvalue, pdbfield)
char *pvalue;
DBFIELD *pdbfield;

getsf(pvalue, pscrfield)
char *pvalue;
SCRFIELD *pscrfield;

putsf(pvalue, pscrfield)
char *pvalue;
SCRFIELD *pscrfield;

Description

Given the pointer to a table buffer field structure, *pdbfield*, **getdf** places the value of the field at the address *pvalue*.

Putdf puts the value pointed to by *pvalue* into the table buffer field specified by *pdbfield*.

Given the pointer to a screen buffer field structure, *pscrfield*, **getsf** places the value of the field at the address *pvalue*.

Putsf puts the value pointed to by *pvalue* into the screen buffer field specified by *pscrfield*.

Note that money fields are stored in units of cents (i.e. 50 dollars is stored as 5000 cents).

Return Value

None.

Name

getdhp - get the pointer to a table header

getdip - get the pointer to a table index

getdfp - get a table field pointer

getshp - get the pointer to a screen form buffer

getsfp - get a field pointer of a screen

Syntax

DBHEAD *getdhp(name)
char *name;

DBINDEX *getdip(pdbhead, name)
DBHEAD *pdbhead;
char *name;

DBFIELD *getdfp(pdbhead, name)
DBHEAD *pdbhead;
char *name;

SCRHEAD *getshp(scrname)
char *scrname;

SCRFIELD *getsfp(pscrhead, name)
SCRHEAD *pscrhead;
char *name;

Description

Given the *name* of a table, **getdhp** returns a pointer to its table buffer structure. If you have allocated memory for your own table buffers, you must have loaded *name* with **fmload**(**T**) to then find it with **getdhp**.

Given the *name* of an index in the table *pdbhead*, **getdip** returns a pointer to the index structure for a table.

Given the *name* of a field in the table buffer *pdbhead*, **getdfp** returns a pointer to the field structure in the table buffer.

Given the *scrname* of a screen (the label of a **SCREEN** section in a **.flx** file), **getshp** returns a pointer to its screen buffer structure. **Getshp** looks among the screens of the currently loaded **.pic** file. Prepend an **a** in front of *scrname* to get the pointer to the array portion of a screen.

Given the *name* of a field in the screen buffer *pscrhead*, **getsfp** returns a pointer to the field structure in the screen buffer.

Return Value

Getdhp returns a table buffer structure pointer if *name* is found, otherwise it returns **NULLDH** (macro in **fxstruct.h**).

Getdip returns a table index structure pointer if *name* is found, otherwise it returns **NULLDI** (macro in **fxstruct.h**).

name is found, otherwise it returns **NULLDF** (macro in **fxstruct.h**).

Getshp returns a screen buffer structure pointer if *scrname* is found, otherwise it returns **NULLSH** (macro in **fxstruct.h**).

Getsfp returns a field structure pointer if *name* is found, otherwise it returns **NULLSF** (macro in **fxstruct.h**).

See Also

fmload

Name

getkey - prompt for a single key

Syntax

getkey( )

Description

Getkey will return the value of the next key press from the keyboard.

Return Value

The key press value is a number between 1 and 255. For values greater than 127, the keystrokes represented are defined in **fxcrt.h** of **$FXDIR/include**.

See Also

prompt

Name

gettime - get the current time in number of seconds since midnight

Syntax

gettime(numtime)
long *numtime;

Description

Gettime gets the current time in terms of the number of seconds since midnight. *Numtime* will point to the time value.

Return Value

None.

See Also

systime

Name

gettoday - get today's date in days since December 31, 1899

Syntax

gettoday(numdate)
long *numdate;

Description

Gettoday gets today's date in terms of the number of days since December 31, 1899. *Numdate* will point to the date value.

Return Value

None.

See Also

sysdate

Name

getxypos - get the CRT row and starting column position of a screen field

Syntax

getxypos(pscrfield, prow, pcol)
SCRFIELD *pscrfield;
int *prow;
int *pcol;

Description

Getxypos will return the physical row, *prow*, and starting column position, *pcol*, for any screen field specified by *pscrfield* belonging to the active screen.

If the screen field is part of an array and the cursor is not positioned in the array portion of the screen, then *prow* will always relate to the first row of the array. If the cursor is within the array, then *prow* will be the row that the cursor addresses.

Limitations

This function can only be used for screen fields belonging to the active screen.

Return Value

None.

Name

gotoxy - position the screen cursor to the specified row and column

Syntax

gotoxy(row, col)
int row, col;

Description

Gotoxy will position the screen display cursor at *row* and *col* of the screen.

Return Value

None.

Name

graphout - draw a horizontal or vertical line

Syntax

graphout(row, col, length, graphmacro)
int row, col;
int length;
int graphmacro;

Description

Graphout draws a line from *row* and *col* on the screen and with the specified *length*. *Graphmacro* defines the kind of character that will be used to draw the line. Horizontal lines are drawn where *graphmacro* is (macros defined in **fxcrt.h**):

GL_THL	single line for top borders.
GD_THL	double line for top borders.
GR_THL	inverse bar for top borders.
GL_BHL	single line for bottom borders.
GD_BHL	double line for bottom borders.
GR_BHL	inverse bar for bottom borders.

Vertical lines are drawn where *graphmacro* is:

GL_LVL	single line for left borders.
GD_LVL	double line for left borders.
GR_LVL	inverse bar for left borders.
GL_RVL	single line for right borders.
GD_RVL	double line for right borders.
GR_RVL	inverse bar for right borders.

Return Value

None.

See Also

boxline

Name

inyesno - prompt for a Y or N response

Syntax

inyesno(row, col, msg)
int row;
int col;
char *msg;

Description

Inyesno displays *msg* at *row* and *col* of the screen display and prompts the user for **Y** or **N** input.

Return Value

See Also

prompt

Name

isempty - test for a 0 or null value in a field

Syntax

isempty(pfield)
FIELD *pfield;

isnull(pfield)
FIELD *pfield;

iszero(pfield)
FIELD *pfield;

Description

Isempty tests if the value in the field specified by *pfield* is 0 or null.

Isnull tests if the value in the field is null.

Iszero tests if the value in the field is 0.

Example

```
if (iszero( @jmaster.jm_rono))
```

Return Value

0 Field does not have the value being tested for.
1 Field has the value being tested for.

Name

ismod - test if a field has been modified

Syntax

ismodfld(pfield)
FIELD *pfield;

ismodrng(pfield1, pfield2)
FIELD *pfield1, *pfield2;

Description

Ismodfld tests if the value in the field specified by *pfield* has been modified.

Ismodrng tests if the value in any field from *pfield1* to *pfield2* has been modified.

Return Value

0 Field has not been modified.
1 Field has been modified.

Name

keychglabel - change a function key label

Syntax

keychglabel(key, label)
int key;
char *label;

Description

Keychglabel will change a function key label as it appears at the bottom of a data entry screen. *Key* is the function key's macro definition as defined in **fxcrt.h** in **$FXDIR/include** (the list starts with **SAVEKEY**). *Label* is the label that will appear for the function key. The label must be four characters or less and be a static variable.

This function is called in the screen's **beforesubsection** or **abeforesubsection** userexits. The label will automatically revert back to the default label upon exiting the screen subsection.

Return Value

None.

Name

lookup - look up in a table based on the specified index

Syntax

lookup(pdbhead, pdbindex)
DBHEAD *pdbhead;
DBINDEX *pdbindex;

Description

Lookup attempts to retrieve a record from the table specified by *pdbhead* accessed by the index specified by *pdbindex*. The values for the index fields must be first assigned to the appropriate fields of the *pdbhead* table buffer.

Return Value

- -1 Table lookup failed. **Iserrno** is set with the specific error code (see **error (T))**.
- 0 Record found with the specified index value.

Name

message - display a message on the screen

Syntax

message(row, col, msg, attr)
int row, col, attr;
char *msg;

msgerr(msg)
char *msg;

msggerr(msg)
char *msg;

msgstat(msg)
char *msg;

msgwait(msg)
char *msg;

msgcomment(msg)
char *msg;

msgfunc(msg)
char *msg;

msgnfunc( )

Description

Message clears a line and displays the message *msg* on the screen at *row*, *col* screen coordinates with attribute *attr*. The possible macro definitions for *attr* are (defined in **fxcrt.h**): **NORMAL**, **BLINK**, **UNDERLINE**, **REVERSE**, **REVBLINK**, **DIMREVERSE**, and **DIM**.

Msgerr places *msg* at line 23 of the screen and rings the bell. **Msgerr** does not require a carriage return to continue.

Msggerr places *msg* at line 23 of the screen, rings the bell, and waits for a carriage return to continue.

Msgstat displays *msg* on the data entry screen's status line, line 23.

Msgwait displays a blinking *msg* on line 23 and asks the user to "please wait".

Msgcomment displays *msg* on the data entry screen's comment line, line 21.

Msgfunc displays *msg* to the function key label line, line 23. The message becomes the new function key label unless *msg* is the global **funcmbuf**.

Msgnfunc displays the function key numbers line, line 22: **F1 F2 F3** ...

See Also

show

Name

mode - table access modes

Description

A table is opened with **fmbuild**(**T**) or **fmopen**(**T**) in one of these modes defined in **fxisam.h**:

ISINPUT	opens the table for reading only.
ISOUTPUT	opens the table for writing only.
ISINOUT	opens the table for reading and writing.

One of these locking parameters is added to the mode:

ISEXCLLOCK	No other process will be allowed to access the table until the table is closed.
ISMANULOCK	A record of the table is locked with a subsequent **fmfind** or **fmread** call where **ISLOCK** is added to the *mode* parameter of the call. A call to **fmrelease** then unlocks all record locked in this way.
ISAUTOLOCK	A record is automatically locked with a **fmfind** or **fmread** and is released with the next table access. Only one record per table can be locked at one time in this manner.

Example

```
fmopen( pdbhead, ISINOUT+ISMANULOCK);
```

See Also

fmbuild, fmopen, fmfind, fmread

Name

modrng - change the modify flag for a range of screen fields

Syntax

modonrng(pscrfirst, pscrlast)
SCRFIELD *pscrfirst;
SCRFIELD *pscrlast;

modoffrng(pscrfirst, pscrlast)
SCRFIELD *pscrfirst;
SCRFIELD *pscrlast;

Description

Modonrng turns on the modify flag in the overall screen buffer and for all the fields within the range of *pscrfirst* thru *pscrlast*.

Modoffrng turns off the modify flag in the overall screen buffer as well as for all fields of the *pscrfirst*/*pscrlast* range.

Return Value

None.

Name

move - copy data from one field to another

Syntax

move(pfield1, pfield2)
FIELD *pfield1, *pfield2;

Description

Move copies the contents of *pfield1* to *pfield2*. This function will convert data types if necessary. Also, one of the arguments may be a C language variable provided it is of the same type and length. The source and the destination can be either a screen field, a database field, or a C variable. However, you cannot use it to move from a C variable to a C variable.

Examples

There are several ways to move values into a screen or table field via user functions. A summary of the various ways are shown below:

```
$table.fieldname = 14;
```

or

```
cnum = 14;
$table.fieldname = cnum;
```

or

```
cnum = 14;
move( &cnum, @table.fieldname);
```

The first is more efficient when dealing with numerics. However, when dealing with character strings move is the proper method to use.

```
move( "teststring", @table.fieldname);
```

Note that money fields are stored in units of cents.

Limitation

Assignment to a screen **array** variable can only be done when the screen is active (currently displayed).

Return Value

None.

See Also

compare, buftostr

Name

nodisplay - set the nodisplay flag for a screen field

Syntax

nodisplay(pscrfield)
SCRFIELD *pscrfield;

unnodisplay(pscrfield)
SCRFIELD *pscrfield;

Description

Nodisplay will set the nodisplay flag on for the screen field specified by *pscrfield*. Anything written to this field thereafter will not appear on the screen.

Unnodisplay will set the nodisplay flag off for the screen field. Anything written to this field thereafter will appear on the screen.

Return Value

None.

Name

nolookup - set the nolookup flag for a screen field

Syntax

nolookup(pscrfield, type)
SCRFIELD *pscrfield;
int type;

Description

Nolookup turns off the look-up process prior to the **aselect**, **abeforedisplay**, and **abeforerow** userexits. Normally look-ups for a screen field (with the **lookup** attribute) are done prior to each of these userexits. If this is not necessary then you can improve the speed and efficiency of the system by turning off the look-up process. A *type* value will turn off look-ups for a specific userexit (the macros are defined in **fxcon.h**):

type	**Userexit effected**
----	----------
LK_SELFUNC	**aselect**
LK_DSPFUNC	**abeforedisplay**
LK_BEFROW	**abeforerow**

You may also control when looked up tables are opened and closed. By or'ing the macro LK_NOOPEN with *type*, the looked up table will be opened and closed as needed rather than be left open. The **nolookup** function should be called in your **abeforesubsection** userexit.

Return Value

None.

Name

page - display all screen literals

Syntax

page(pscrhead)
DBHEAD *pscrhead;

Description

Page will display all the literals for the screen specified by **pscrhead**.

Return Value

None.

See Also

repaint

Name

prompt - prompt for data from any position on the screen

Syntax

prompt(row, col, length, dec, type, attr, fmt, buffer)
int row;
int col;
int length;
int dec;
int type;
int attr;
int fmt;
char *buffer;

Description

Prompt will prompt for *length* characters of input into *buffer* at *row* and *col* of the screen display.

Dec is a flag indicating that a fractional part of a number input is permitted. If *dec* is a value of 0, then a decimal point is illegal input.

Type is the data type being prompted for (these macros are defined in **fxtypes.h**):

CHARTYPE	character string
SHORTTYPE	short integer
INTTYPE	integer
LONGTYPE	long integer
DOUBLETYPE	double-precision floating point number
DECTYPE	decimal structure
SERIALTYPE	serial long integer
DATETYPE	date
MONEYTYPE	money
TIMETYPE	AM/PM style time
MTIMETYPE	military time

The *attr* argument defines input requirements. The possible macro definitions for *attr* are defined in **fxcon.h**:

A_NODISPLAY	field is not displayed.
A_AUTONEXT	Automatic RETURN key when field is filled.
A_ALNUM	allow alphabetic characters or numeric.
A_ALPHA	allow alphabetic strings only.
A_NUMERIC	allow numeric field only.
A_DNSHIFT	map input to lower case.
A_UPSHIFT	map input to upper case.
A_REVERSE	use reverse video attribute for field.

Fmt is a format flag. The format flags that are meaningful to **prompt** are defined in **fxcon.h** and described under the function **buftostr**. These flags may be or'ed together.

F_TRUNCATE	For **CHARTYPE** fields blank any other characters in the prompt field with input of the first character into the field.
F_PHONE	Input must be legal characters of a phone number.

Buffer must be at least one byte longer than *length*.

Return Value

None.

Name

putkey - display a single character on the screen

Syntax

putkey(row, col, key)
int row, col, key;

Description

Putkey displays the character *key* at *row* and *col* of the screen display.

Return Value

None.

Name

repaint - repaint the active screen and any background screens.

Syntax

repaint( )

Description

Repaint redraws the literals of the active **SCREENFLEX** screen and any background screens if the active screen is a pop-up.

Return Value

None.

See Also

page

Name

rpt - manipulate the **reportflex** output

Syntax

rptformfeed()

rptgetline()

rptlneed(n)
int n;

rptline(buffer)
char *buffer;

long rptpageno()

rptposition(n)
int n;

rptprint()

rptsection(pscrhead)
SCRHEAD *pscrhead;

long rptreccount()

Description

Rptformfeed starts newpage for report.

Rptgetline returns current report line number.

Rptlneed requests *n* number of lines to the bottom of the report page. If that number of lines is not available, then the next output of the report will appear on a new page. Use **rptlneed** if you want a certain number of lines not to be broken across pages, for example, a header being printed on one page and the detail to that header beginning on the next page. The *n* number of lines should include any lines taken up by a page footer and bottom margin.

Rptline prints *buffer* to the output file of a report.

Rptpageno returns current page number. Rptpageno returns value as a long.

Rptposition skips to report line number *n*.

Rptprint outputs a record formatted according to the REPORT section. **Rptprint** may only be used in the **section** userexit of the SELECT section immediately preceding the REPORT section.

Rptsection outputs a complete REPORT section. **Rptsection** may be used in the the **beforeprint() and afterprint()** userexits.

Rptreccount returns record count for primary file. Rptreccount returns value as a long.

Limitations

All these functions are only for use in **REPORTFLEX** userexits.

Return Value

None.

Name

rtimestr - convert an internal time format to a string

Syntax

rtimestr(secs, stime, type)
long secs;
char *stime;
int type;

rstrtime(stime, secs, type)
char *stime;
long *secs;
int type;

Description

Rtimestr converts the time in internal format, *secs*, to a display format string, *stime*. Internal format is the number of seconds since midnight, where midnight itself is 86400. A 0 value for *secs* results in blank *stime*. *Type* is **TIMETYPE** or **MTIMETYPE** (military time) which defines the format of *stime* (these macros are defined in **fxtypes.h**).

Rstrtime also converts a display format string, *stime*, to the time in internal format. The resultant internal time is pointed to by *secs*. *Type* is one of the same two macros used by **rtimestr**. A blank input string will zero the internal time.

Return Value

-1 *Secs* out of range (**rtimestr**), bad *stime* format (**rstrtime**), or illegal *type*.
0 Conversion successful.

Name

sclr - clear fields in the screen buffer

Syntax

sclrrec(pscrhead)
SCRHEAD *pscrhead;

sclrfld(pscrfield)
SCRFIELD *pscrfield;

sclrrng(pscrfirst, pscrlast)
SCRFIELD *pscrfirst, *pscrlast;

Description

Sclrrec clears all fields in the screen buffer specified by *pscrhead*.

Sclrfld clears the single screen buffer field specified by *pscrfield*.

Sclrrng clears each field of a range of screen buffer fields from *pscrfirst* to *pscrlast*.

Return Value

None.

See Also

dclr, tclr

Name

scrollpage - scroll the array portion of the screen

Syntax

scrollpage(mode)
int mode;

Description

Scrollpage will scroll the array portion of a screen form based on the value of *mode* (these macros are defined in **fxform.h**):

R_UPLINE scroll up 1 line.
R_FWRD scroll forward 1 page.
R_DOWNLINE scroll down 1 line.
R_BACK scroll back 1 page.
R_FIRST scroll to first page.
R_LAST scroll to last page.
R_MREPAINT scroll current page; used to repaint the array portion of a screen after temporarily overwriting it with a another screen or other literals. The data will be repainted from memory.
R_DREPAINT scroll current page; used to repaint the array portion of a screen after temporarily overwriting it with a another screen or other literals. The data will be repainted by rereading the disk.

Return Value

None.

Name

setcursor - turn the screen cursor on or off

Syntax

setcursor(mode)
int mode;

Description

Setcursor turns the screen cursor on when *mode* is **ON** and turns it off when *mode* is **OFF** (these macros are defined in **fxcon.h**).

Return Value

None.

Name

setnull - set the value of field to null

Syntax

setnull(pfield)
FIELD *pfield;

setzero(pfield)
PFIELD *pfield;

Description

Setnull nulls the value of the field specified by *pfield*.

Setzero sets the value of the field to 0.

Return Value

None.

Name

sfswap - swap two screen fields

Syntax

sfswap(pscrfield1, pscrfield2)
SCRFIELD *pscrfield1, *pscrfield2;

Description

Sfswap will dynamically swap two screen fields. When swapped, *pscrfield1* will take on all of the field attributes of *pscrfield2* but will retain *pscrfield1*'s original row and column placement on the screen.

Return Value

None.

Name

show - display a message on the screen

Syntax

show(msg, attr)
char *msg;
int attr;

bshow(msg, attr)
char *msg;
int attr;

showxy(row, col, msg, attr)
int row, col;
char *msg;
int attr;

bshowxy(row, col, msg, attr)
int row, col;
char *msg;
int attr;

Description

Show and **bshow** display the message *msg* at the current x, y coordinate.

Showxy and **bshowxy** display *msg* at the given *row*, *col* coordinate.

Bshow and **bshowxy** only buffer the message, which is displayed when **bflush** is called.

The possible macro definitions for *attr* are defined in **fxcrt.h**: **NORMAL**, **BLINK**, **UNDERLINE**, **REVERSE**, **REVBLINK**, **DIMREVERSE**, and **DIM**.

Row may have a value of 0 to 23. *Col* may have a value of 0 to 79. For portablility on UNIX, be careful not to use column 79 for attributed text because some terminals require an additional column for the attribute.

Displaying a message using any of these functions does not first clear the display line.

Return Value

None.

See Also

message, bflush

Name

skip - mark a field so it is skipped during data entry

Syntax

skip(pscrfield)
SCRFIELD *pscrfield;

noskip(pscrfield)
SCRFIELD *pscrfield;

Description

Skip marks a screen field specified by *pscrfield* to be skipped during data entry.

Noskip marks *pscrfield* not to be skipped.

Return Value

None.

Name

skipto - specify the next screen field to take input

Syntax

skipto(pscrfield)
SCRFIELD *pscrfield;

Description

Skipto specifies that *pscrfield* is the next screen field to take input after data entry to the current one.

If the user is cursoring back a field or has pressed **JUMP**, **ZOOM**, or **HL** (help), the **skipto** will not take effect.

Limitations

Skipto should be called from the **afterfield** userexit.

Return Value

None.

Name

smap - copy fields from the table buffer to the screen buffer

Syntax

smaprec(pdbhead, pscrhead)
DBHEAD *pdbhead;
SCRHEAD *pscrhead;

smapfld(pdbhead, pscrfield)
DBHEAD *pdbhead;
SCRFIELD *pscrfield;

smaprng(pdbhead, pscrfield1, pscrfield2)
DBHEAD *pdbhead;
SCRFIELD *pscrfield1, *pscrfield2;

Description

Smaprec copies the data of any fields from the table buffer specified by *pdbhead* to the corresponding fields of the screen buffer specified by *pscrhead*.

Smapfld copies the data from the corresponding field of the table buffer to the screen buffer field specified by *pscrfield*.

Smaprng copies the data from the corresponding fields of the table buffer to a range of screen buffer fields from *pscrfield1* to *pscrfield2*.

Return Value

None.

See Also

dmap, tmap

Name

str - operations on strings

Syntax

```
char *strscan( s)
register char *s;

char *strcompress( s)
char *s;

char *strtrim( s, c)
char *s;
int c;

char *strltrim( s, c)
char *s;
int c;

strcenter( s, len)
char *s;
int len;

char *strfind( s, subs, comp)
register char *s;
char *subs;
int (*comp)( );
```

Description

Strscan deletes leading white space in the string *s*.

Strcompress deletes leading and trailing white space from a string *s*.

Strtrim trims the character *c* from the right of the string *s*.

Strltrim trims the character *c* from the left of the string *s*.

Strcenter centers the string *s* within the first *len* bytes of *s*. The size of string *s* must be greater than or equal to *len*, and *len* must not be greater than 255.

Strfind will attempt to find the substring *subs* in the string *s*. *Comp* is a pointer to the comparison function, e.g. **strncmp**.

Return Value

All functions, except **strcenter**, return a pointer to the resultant string. **Strfind** will return the **NULLCHAR** (defined in **fxcon.h**) if *subs* is not found in *s*.

Name

sysdate - get the system date

Syntax

sysdate(datebuf)
char *datebuf

Description

Sysdate formats the current system date into a buffer pointed to by *datebuf*. The format of the date will be according to the **FXDATE** environment variable, and if that variable is undefined, the format will be *MM*/*DD*/*YY*.

Return Value

None.

See Also

gettoday

Name

systime - get the system time

Syntax

systime(timebuf)
char *timebuf;

Description

Systime formats the current system time into a buffer pointed to by *datebuf* in military format: *hh*:*mm*:*ss*.

Return Value

None.

See Also

gettime

Name

tclr - clear screen fields on the display

Syntax

tclrrec(pscrhead)
SCRHEAD *pscrhead;

tclrall(pscrhead)
SCRHEAD *pscrhead;

tclrfld(pscrfield)
SCRFIELD *pscrfield;

tclrrng(pscrfirst, pscrlast)
SCRFIELD *pscrfirst;
SCRFIELD *pscrlast;

Description

Tclrrec clears all screen fields of the screen specified by *pscrhead*, except in the case of an array screen where only the fields of the current array row are cleared.

Tclrall clears all screen fields of the screen specified by *pscrhead*. In the case of an array screen, all fields of every array row are cleared.

Tclrfld clears the screen field specified by *pscrfield*.

Tclrrng clears the range of screen fields from *pscrfirst* to *pscrlast*.

Return Value

None.

See Also

clr, dclr, sclr

Name

tmap - display data to screen fields

Syntax

tmaprec(pscrhead)
SCRHEAD *pscrhead;

tmapfld(pscrfield)
SCRFIELD *pscrfield;

tmaprng(pscrfirst, pscrlast)
SCRFIELD *pscrfirst;
SCRFIELD *pscrlast;

Description

Tmaprec displays all the data of the fields of the screen buffer specified by *pscrhead*.

Tmapfld displays the data of a single screen field specified by *pscrfield*.

Tmaprng displays the data of a range of screen fields from *pscrfirst* to *pscrlast*.

Return Value

None.

See Also

dmap, smap
ma

INDEX