CSG IAS

Tutorial & Reference Manual

Clearbrook Software Group

CLEARBROOK SOFTWARE GROUP
INFORMATION MANAGEMENT SYSTEM

(CSG IMS)

Release B, January 1, 1986

By Paul Kehler and Paul Goertz

Documentation by Clarence Martens

Clearbrook Software Group
PO Box 8000-499
Abbotsford, BC
CANADA V2S 6H1

Clearbrook Software Group
446 Harrison Street
PO Box 8000-499, Sumas, WA
USA 98295-8000

Phone: (604)853-9118

TABLE OF CONTENTS

FOREWORD

Congratulations on your purchase of the Clearbrook Software Group Information Management System (CSG IMS). CSG IMS is an extremely powerful yet easy to use database manager that includes all the necessary tools to create powerful business software. It has both relational and network capabilities.

This documentation is meant for two kinds of users. For experienced programmers the tutorial section should be read to get the "feel" of IMS, and then the reference section should be studied to learn the specifics of each IMS statement.

For beginners, the Glossary is a good place to start. The tutorial is next; this is a step by step section showing how simple it really is to create a powerful IMS program. It does not go through each detail of IMS, rather it is a learning by example guide. Once through this, the reference section should be studied to learn all the details of IMS.

But before you go any further you should fill out and return the license agreement and then turn to **appendix A** for instructions on how to install IMS.

Note that throughout the documentation, OS9 is a registered trademark of Microware Systems Corp. and Motorala.

CSG IMS is a trademark of Clearbrook Software Group Inc.

blank

GLOSSARY

A dictionary of words used in CSG IMS documentation.

assignment
To give a variable or field the value of an expression. Assignments have the variable name or field name on the left side, an equals symbol (= or **EQ**) in the middle and an expression on the right side.

branch
To change the execution order of a IMS program. Branch is a general term to describe all the commands in IMS that change the consecutive order of execution of an IMS program.

byte
A unit of storage on disk or in memory. A character is stored in one byte.

character
A letter, digit, punctuation mark, etc. "The name" has 8 characters between the quotes.

compile
To change a text file containing MODULE(S) to a file which can be executed by the IMS INTERPRETER.

control character
A character which cannot be displayed. On terminals, control characters are often used to perform functions, such as clearing the screen, or scrolling the text on the screen up one line. Control characters can also be typed on the keyboard to control the operation of a program. To type a control character, hold the key labeled CTRL down while you type a letter key.

cursor
An indicator on the screen ot a terminal which occupies one character. It is where the next character will be printed, and when it is, the cursor will shift right by one character. It is often seen as an underline, or a reverse video block, and sometimes flashes.

data file
A number of records on disk. For example, the data file called **APPLICANTS** could be the name or a file that contains many records, each containing information about

prospective employees. When the word **file** is used by itself it refers to a data file.

declaration

A declaration is any IMS statement that declares that a named variable exists and of what type it is. For example:

```
REAL amount
```

declares the variable **amount** to be of type REAL.

default

A default is the value that is assumed when no value is stated. For example, the default left and right margins for reports are at the first and eightieth columns. That means that unless the margins are explicitly changed, reports will produce output with these margins.

execution

The word used to describe IMS running or executing your module. When a particular statement is said to be executed, that means the statement has given IMS an instruction to accomplish and IMS is doing it.

expression

An expression is a value, possibly containing functions, operators, numbers, etc. **3** is an expression with the value of 3, **3+5** is another with value 8, **LEFT$("this is the text",4)** is another with value "this".

field

An item of information in a record. For example: **NAME_OF_APPLICANT** could be a field in a record of a file called **APPLICANTS**.

file descriptor

This is a text file made by the user telling IMS the various fields and keys in the file and the masks that go with each. This file is used to create the data file.

function

A function in IMS is a statement that returns a value. Functions can be printed, used in an expression, or assigned to a variable. For example, **MAX** is a function that returns the largest value in a series of values, and **PRINT** is a statement that outputs a value:

```
PRINT MAX (10,20,-34)
```

would output 20.

index file

An index file differs from a data file in that the information for the key values of the data are stored there. The name of the index file is similar to the name of the data file the index is for. A data file has an extension of

.ide and the index file has an extension of **.iin.** When the word **file** is used by itself it refers to a data file.

input
A general word to describe data coming into the computer. Typing on the computer keyboard is a form of input.

interpret
To execute a complied MODULE.

key
An expression of field values that species an order for records in a data file. For example, the employee number field in an hours worked file would probably be a key, meaning that searches, listings, and operations on the file could be done in the order of the employee number.

loop
A group of statements that execute for a number of times until another statement is used to stop them. Loops in IMS are marked by LOOP ... ENDLOOP, REPEAT ... UNTIL, and WHILE ... ENDWHILE statements.

mask
The way the user specifies the format the information will be entered into the field and displayed from the field. Masks are set up in the file descriptor and can be changed during execution of a module.

module
An IMS program, marked by the MODULE statement at the beginning and an optional END at the end.

null TEXT
No characters. The null TEXT value would be the value in between the quotes "", which is nothing at all. Note that spaces are different from the null value.

output
To send data out from the computer. The computer sending a message to the screen is an example of output.

paint
To make a form on the screen. **Paint** refers to the process in which the user specifies titles and placement of fields on a screen or report form.

pathlist
The name of a file including the hierarchical directory information. In OS9 an example is **/d0/data/accounts.**

program
A sequence of instruction to the computer. In IMS, a program is also called a module.

record
An individual grouping of data. A record relates information about a particular person or item into one unit. For example, in a mail list a record would be the mailing information for a person you mail to ,ie., the name and address ot that person.

relational
1. The ability of a data base program to link related data files together so that the total sum of information can be handled in a simple manner. For example, a file containing invoices can be related to the file containing vendor information, so that the information about the vendor and the invoices can be handled as one unit. IMS is a relational language.
2. A symbol such as < (less than) or > (greater than) that relate one value to another. Relationals return a true or false ındicating if the first value is truly related in that way to the second value.

reserved word
All the words of IMS statements are known as reserved words. This includes IF, NEXT, FIND, SCAN, etc. (they are normally in all capital letters). See the reference manual for a full list.

statement
A statement is just a single line in an IMS module, comprising a single instruction.

subroutine
A module or a group of statements inside a module that performs a task useful in a variety of instances. Typically a GOSUB or CALL instruction is used to execute the subroutine.

variable
A name of a storage location. A variable holds a value that can be change by assignment.

SPECIFICATIONS

```
Maximum data file size.....OS Limited *
Maximum number of records per data file.....OS Limited
Maximum number of fields per record.....memory limited **
Maximum number of bytes per record.....memory limited
Minimum number of bytes per record.....5
Maximum number of OPEN files.....5 or 6 ***
Maximum number of keys per file.....127
Maximum length of a single field.....memory limited
Maximum length of a field or variable name.....255
Maximum number of lines in a module.....memory limited
```

NOTE:

* OS Limited: Limited by the disk capacity and operating system.

** Memory Limited: In OS9 approximately 20K is available for program modules and data.

*** Most OS9 systems allow 16 open paths. Three paths are used for standard input, output and error. Another path is used for the error message file and two may be used for alternative input and output paths. Each OPEN statement opens a data file and an index file.

blank

SYSTEM REQUIREMENTS

The following are the basic system requirements for IMS.

Operating System
OS9/6809 LII

Ram Memory
128K or more

Mass Storage
2 double sided double density drives of 250K or more each. Hard disk drives are recommended for more involved applications.

CRT Terminal
-absolute cursor addressing
-nonembedded video attribute(s) (at least one), for example: half intensity or reverse video.
-clear screen
-clear to end of line

Printer
ASCII printer with 80 or more columns which recognizes the ASCII formfeed character.

blank

UPDATE POLICY

Clearbrook Software Group will provide free updates for a period of one year after the purchase of the software. After this period, updates will be available for $15US ($20US overseas).

To receive your update, you must send your original disk (or your most recent update) to Clearbrook Software Group. We will then send you our latest update. Please check with us first to determine if an update is available.

The licence agreement enclosed with this manual serves two purposes. When returned to us, it will place you on our mailing list. We will then automatically keep you informed of when updates are available. The signed licence is an agreement that the licencee and Clearbrook Software Group will abide by the terms of the licence.

TUTORIAL

INFORMATION MANAGEMENT SYSTEM

Clearbrook Software Group

CLEARBROOK SOFTWARE GROUP
INFORMATION MANAGEMENT SYSTEM

TUTORIAL

Release B
January 1, 1986

TABLE OF CONTENTS

INTRODUCTION

This tutorial will introduce you to the various aspects of the IMS application development environment. You will learn how to "run" IMS programs, create data files, design a screen input form and do reports. You will also learn how to write and modify your own programs and to work in the interactive environment.

The tutorial section is not intended to cover all aspects of IMS. The reference section gives you an organized description of all statements and functions available for writing programs. The appendices have more detailed information on the use of IMS.

Before you get started with the tutorial, you should make sure you have installed IMS on your computer. Refer to Appendix A for instructions to do this.

Another thing to know is how data is expected to be entered. Text is typed in the normal manner. The key labelled DEL is used to delete the previously typed character. The left and right arrow keys are used to move the cursor. When you have entered your data and ensured that it is correct, you will usually need to press the RETURN or ENTER key (on some keyboards this key is labelled with a large bent arrow) to indicate that you are satisfied with what you have typed. This signals the computer to interpret what you have told it to do.

NOTE: The files used in the tutorial are found in the **IMS** directory of your diskette. Make sure this is your current data directory when you do the following lessons.

Now it is time to start the tutorial, enjoy!

Directory: /DO/IMS Date: January 18, 1986

CSG IMS Executive

1. Editor
2. Generate a data file
3. Paint a screen form
4. Describe a report format
5. Compile module
6. Execute a compiled module
7. Interactive environment
8. Change working directory
9. Pass a command to the operating system
10. Quit

Your choice:

CSG Information Management System
Version 1.0, Serial number 000000

LESSON 1

Getting Started

Objectives:

To learn
- the main menu options
- the interactive mode
- how to maintain a file
- the file commands
  - OPEN
  - LIST STRUCTURE
  - LIST
  - PRINT
  - DELETE
  - SET PRINT

In this lesson you will learn how to use an IMS module that maintains the data in a file and how to use the interactive mode to make queries about the data in the file.

From the operating system prompt, type **IMS** and press the RETURN or ENTER key. When IMS is ready you get the screen called "main menu", which is reproduced on the opposite page. It is a list of the various options you can choose in IMS. These options are selected by typing in their respective number and pressing the RETURN or ENTER key. We will see them in operation in the various lessons of the tutorial.

1. Editor

Selecting **1** will choose the editor option. This will give the prompt:

File to edit:

You will also see above the prompt line a menu of file names. These are the text files; files that end in **.imo** are IMS modules, files that end in **.ide** are IMS data descriptors. **.imo** and **.ide** are known as file extensions, and these are the recommended file extensions for these kinds of files. To edit one of these files simply type in its name, or you can type in a new text file name, but don't forget to include the extension as part of the name.

Directory: /DO/IMS Date: January 18, 1986

CSG IMS Executive

1. Editor
2. Generate a data file
3. Paint a screen form
4. Describe a report format
5. Compile module
6. Execute a compiled module
7. Interactive environment
8. Change working directory
9. Pass a command to the operating system
10. Quit

Your choice:

CSG Information Management System
Version 1.0, Serial number 000000

2. **Generate a data file**
This will give the prompt:

Name of file descriptor:

At this point you can type in the name of a file from the list above the prompt. Not all the files listed are data descriptor files. We recommend that you put an extension of **.ide** at the end of a file name so you can identify files from which you can generate data files. IMS will make one or more data files based on the information in the file descriptor.

3. **Paint a screen form**
This option will give the prompt:

Data base file(s):

and wait for you to type in the names of the data base files for which you wish to create or edit a screen form. Above the prompt, available data base file names will be displayed. This option will let you design the form, and can generate an IMS module to maintain one of the files used in this form.

4. **Describe a report format**
This option will prompt:

Data base file(s):

and wait for you to type in the names of the data base files for which you wish to create or edit a report form. Available data base file names will be displayed above the prompt. This option will let you design the report format and generate an IMS module to do the report.

5. **Compile module**
With this option you get the prompt:

Source file to compile:

Answer the prompt with the name of a file containing an IMS module. Possible file names are listed above the prompt (not all of the files listed will be IMS modules). In IMS you type in the module using the text editor, but you cannot execute the module until you compile it first. Compiling means that the words and phrases you typed into the editor are converted into instructions that IMS can understand and execute.

MAIL LIST FOR EVERYONE
==========

NAME: Jim Smith
ADDRESS: 2341 West Hauser Street
Anytown, Anywhere
COUNTRY: BANZAI
POSTAL CODE: V2E 2W2
COMMENT: Buckaroo

FILE: maillist KEY: NOKEY FORM: maillist RECORD #7

6. **Execute a compiled module**
   This prompts:

   Module to execute:

   Answer with the file name of a compiled module. Before any module can be executed it must be compiled by option 5.

7. **Interactive environment**
   This allows you to make queries on the file information and get instant results.

8. **Change working directory**
   If you want to work with files in a different data directory choose this option. A prompt will be given:

   New data directory:

   to let you type in the new directory name.

9. **Pass a command to the operating system**
   Choosing this option gives the prompt:

   Shell command:

   Here you can type in any operating system command. Note, on OS9 systems you can press RETURN or ENTER to get a new shell.

10. **Quit**
    Choosing this option returns you to the operating system.

Press 6 to execute an IMS module. IMS will prompt:

Module to execute:

You can now type in the name of the file to execute; in this lesson type in **maillist** and press the RETURN or ENTER key. **maillist** is a module that was created previously for this lesson.

After a few moments the screen will show a form like that on the opposite page. Go ahead, type a name and address into the form. You have to press the ENTER of RETRUN key after entering each line. Notice that the Postal Code has the format of letter number letter space number letter number; an example would be "V2T 1E6".

MAIL LIST FOR EVERYONE
==========

NAME: Jim Smith
ADDRESS: 2341 West Hauser Street
Anytown, Anywhere
COUNTRY: BANZAI
POSTAL CODE: V2E 2W2
COMMENT: Buckaroo

SELECT: Insert Update Clear Delete First Last Next Previous Key Search Quit

After filling the form, press the ESC key. This will cause a small menu to be printed on the bottom of the screen as follows:

Insert Update Clear Delete First Last Next Previous Key Search Quit

These are 11 options that can be selected by pressing the first letter of the word. Each is explained below:

Insert
This will add the data as it appears on the screen to the disk file. The data on the screen form is actually one unit, and is called a **record**. Inserting a record in fact creates a new record in the file.

Update
This will update the current disk file record with the data currently on the screen. Use Search, First, Last, Next or Previous to find a record before editting and updating it.

Clear
This will clear or blank out the fields on the screen. Note, the data in the file will not change unless you Update after the Clear. So, if the screen has:

```
          NAME:   Jim Smith
       ADDRESS:   2341 West Hauser Street
                  Anytown, Anywhere
       COUNTRY:   BANZAI
   POSTAL CODE:   V2E 2W2
       COMMENT:   Buckaroo
```

pressing **ESC C** will make it appear as:

```
          NAME:
       ADDRESS:

       COUNTRY:
   POSTAL CODE:
       COMMENT:
```

Delete
This will delete the record currently on the screen from the file. In other words, the data will be forgotten.

First
Selecting this option will display the first record in the file. If the file is currently in the NOKEY order, the first record stored in the file is displayed. If the **name** key is selected, the records are ordered alphabetically by the name so the record with the alphabetically first name

```
                MAIL LIST FOR EVERYONE
                ======================

          NAME: Jim Smith
       ADDRESS: 2341 West Hauser Street
                Anytown, Anywhere
       COUNTRY: BANZAI
   POSTAL CODE: V2E 2W2
       COMMENT: Buckaroo


SELECT:  Insert Update Clear Delete First Last Next Previous Key Search Quit
```

will be displayed.

Last

This will display the last record according to the currently selected key.

Next

This will display the next record on the screen form. The current key is used to decide which is the next record.

Previous

This will display the previous record on the screen form.

Key

The Key option will give the prompt:

```
Choose one field:
 * 1 - NOKEY
   2 - name
Selection?
```

Every file is ordered in some manner. In this example, the file can be ordered alphabetically by the **name** field. The keys themselves are given names (this will be seen in lesson 2) and in this example we have a **name** key and a NOKEY key. The NOKEY key has an asterisk beside it to show that currently the file is ordered by that key. The current key order is used for the First, Last, Next and Previous actions and for Searching.

NOKEY is a special key that retreives data in the same order as it is stored in the file. This is similar to the order in which the data was typed in except that deleted records are replaced before records are added at the end of the file. So if in an empty file the data was typed in the order of "Harry Smith", "John Dellert", and "James Mann" then the first record would be "Harry Smith", the next record would be "James Mann", and the last record would be "John Dellert".

There may be several keys per file. The Key option is a convenient way of changing the ordering of the file at any time.

Search

A search will look through the entire file for an approximate match with the data typed into the key field(s). The current key cannot be NOKEY when you do the Search. To search, you type a value into data field(s) which correspond to the current key, then press **ESC** followed by **S**. IMS will search for the first record that matches the key or the next

```
CSG_IMS v1.0
(c) Clearbrook Software, 1985
IMS:
IMS:open'maillist'
IMS:
IMS:list structure
*maillist / maillist.ida contains 8 records, 152 bytes long.
    INDEX                   TYPE      LENGTH
    -----                   ----      ------
    name                    text      30


  FIELD NAME          TYPE    OFFSET LENGTH MASK
  ----------          ----    ------ ------ ----
  name                text    1      30
  address1            text    31     30
  address2            text    61     30
  country             text    91     15     LLLLLLLLLLLLLLL
  postal_code         text    106    6      L#L #L#
  comment             text    112    40

IMS:
```

greater if no exact match is found. The screen will show the record which was found.

For example, suppose the file contained information about "Harry Smith", "James Dellert", and "Jim Anderson". You could then put the cursor onto the **name** field and type in "Jerry Lewis", press the **ESC** key and then the **S** key. IMS would then search through the file looking for the record with the name "Jerry Lewis"; it would not find an exact match and instead it would show the next record - "Jim Anderson".

Quit

Pressing the **ESC** key and then the **Q** key selects this option, which ends execution of the module and returns to the main menu.

Go ahead and try these ESC options. A helpful hint is that if you press the ESC key but don't wish to do an ESC option then simply press ESC again to get out of the menu.

Once you have finished trying out the options press **ESC** and **Q** to quit the module. This will bring you back to the main menu, from where you can choose option 7. This will bring you into the interactive mode. The interactive mode is convenient because it allows you to type IMS file commands and get an immediate response. The first of the file commands we will look at is OPEN.

Before any work can be done with a file, it must first be OPENed. Type in the following statement:

```
OPEN "maillist"
```

Be sure to press RETURN or ENTER at the end of the line. This will open the IMS data file called **maillist** in the disk directory. Now the IMS commands can work on this file. Alternatively, you could type:

```
OPEN "maillist" AS ml
```

which would open the same file as before, but this time the IMS commands use "ml" as the name of this file. This can be a handy way of abbreviating file names to save typing.

In order to see what kind of information our file holds type:

```
LIST STRUCTURE
```

This will give you a list of the keys and fields which are contained in the file. To use information from a file you must use the names of the fields to indicate which informa-

```
IMS:list
Mary Jones                    123 Hidden Rd.                Cloven Hoof, AL
          USA                       Student
Paul Davis                    34 University Drive           Seattle, WA
          USA                       Student
Richard McBride               666 Glitter Ave.              Miami, FL
          USA                       Student
Wimpy                         435 Back Alley                Skidroad
          ANY PLACE                 a moocher
Freeloader Fred               121 Easy Street               New York, NY
          USA                       a moocher
The Sponger                   208-3110 Dreary Lane          Calgary, AL
          CANADA              A0A 0A0 a moocher
Jim Smith                     2341 West Hauser Street       Anytown, Anywhere
          BANZAI              V2E 2W2 Buckaroo
Dave Wilson                   123 Harrison Street           Sumas, WA
          AMERICA                   Student
IMS:

IMS:list all for country<>"USA"
Wimpy                         435 Back Alley                Skidroad
          ANY PLACE                 a moocher
The Sponger                   208-3110 Dreary Lane          Calgary, AL
          CANADA              A0A 0A0 a moocher
Jim Smith                     2341 West Hauser Street       Anytown, Anywhere
          BANZAI              V2E 2W2 Buckaroo
Dave Wilson                   123 Harrison Street           Sumas, WA
          AMERICA                   Student
IMS:list all for country="USA"
Mary Jones                    123 Hidden Rd.                Cloven Hoof, AL
          USA                       Student
Paul Davis                    34 University Drive           Seattle, WA
          USA                       Student
Richard McBride               666 Glitter Ave.              Miami, FL
          USA                       Student
Freeloader Fred               121 Easy Street               New York, NY
          USA                       a moocher
IMS:

IMS:list all for country="USA" and comment="Student"
Mary Jones                    123 Hidden Rd.                Cloven Hoof, AL
          USA                       Student
Paul Davis                    34 University Drive           Seattle, WA
          USA                       Student
Richard McBride               666 Glitter Ave.              Miami, FL
          USA                       Student
IMS:
```

tion is required.

LIST is an IMS file command. It prints out all the records that meet the stated **range specification.** A range specification is a series of conditions that tell IMS what records the file command is to work with. It can be written in many ways, in the simplest form you can type:

```
LIST
```

which will print out all the records. Here the range specification is implied. With the LIST command IMS uses the default specification of ALL the records, when the range specification is not present.

```
LIST ALL
```

is another way of doing a list command, here the range specification is ALL the records. This file command will do exactly what the previous one did.

Another way of doing a LIST statement is in the form:

```
LIST ALL FOR condition
```

condition specifies a test on a field value(s), letting the file command LIST print out the record only if the test passes (ie., the condition is true). For example:

```
LIST ALL FOR country <> "USA"
```

will print out all the records where the country field is not equal to USA. The symbol **<>** means "not equal to" and could instead be written **NE.**

```
LIST ALL FOR country = "USA"
```

This will do the opposite; it will print out all the records with the country field equal to "USA". The symbol = means equal to, and could be written as **EQ** instead.

```
LIST ALL FOR comment="Student" AND country="USA"
```

This is a double condition; it is saying that if the comment field is "Student" AND the country field is "USA" then print out the record. The AND part means that both conditions must be true for the record to be printed out.

```
LIST ALL FOR comment="Student" AND country=
     "USA" PRINT name; " is a student"
```

Note that this is typed in as one line. In fact, all IMS

```
IMS:list all for comment="Student" and country="USA" print name;" is a student"
Mary Jones is a student
Paul Davis is a student
Richard McBride is a student
IMS:
IMS:scan all for country="AMERICA" let country="USA" print name
Dave Wilson
IMS:
IMS:list
Mary Jones                    123 Hidden Rd.                 Cloven Hoof, AL
            USA                     Student
Paul Davis                    34 University Drive            Seattle, WA
            USA                     Student
Richard McBride               666 Glitter Ave.               Miami, FL
            USA                     Student
Wimpy                         435 Back Alley                 Skidroad
            ANY PLACE               a moocher
Freeloader Fred               121 Easy Street                New York, NY
            USA                     a moocher
The Sponger                   208-3110 Dreary Lane           Calgary, AL
            CANADA          A0A 0A0 a moocher
Jim Smith                     2341 West Hauser Street        Anytown, Anywhere
            BANZAI          V2E 2W2 Buckaroo
Dave Wilson                   123 Harrison Street            Sumas, WA
            USA                     Student
IMS:
```

statements are entered as one line. This has the same range specification as the previous example, but this time an **action** has been added. The action is:

```
PRINT name; " is a student"
```

which will print out the name of the person(s) with a comment field of "Student" and country of "USA" followed by " is a student". Note that this LIST will not print out all the fields of the record; since we are specifying the name field to be printed, the the rest of the fields will not be printed. The ";" character is special in a PRINT statement; it says not to print a new line before printing the next data.

An example would be:

```
Mary Jones is a student
Paul Davis is a student
Richard McBride is a student
```

In summary, the effect of LIST is to print out records from the file. It can be followed by a range specification like:

```
ALL
```

or

```
FOR comment="Student"
```

and then an action like:

```
PRINT name
```

Another file command is SCAN. Like most file commands, SCAN has a range specification and an action like LIST, but it can also have a LET action statement. The LET statement is a way of assigning new values to fields. For example:

```
SCAN ALL FOR country="AMERICA" LET country="USA"
     PRINT name
```

This statement will go through all the records and check to see if the country field is "AMERICA". If it is, then the country field is changed to "USA" and the name of the person is printed out.

Another example would be:

```
SCAN ALL FOR country= "CANADA" AND postal_code=
     "V2T-1E6" LET comment="My neighbour" PRINT
     name
```

```
IMS:delete all for comment="a moocher" print name;" is deleted"
Wimpy is deleted
Freeloader Fred is deleted
The Sponger is deleted
IMS:
IMS:list
Mary Jones                    123 Hidden Rd.                 Cloven Hoof, AL
            USA                     Student
Paul Davis                    34 University Drive            Seattle, WA
            USA                     Student
Richard McBride               666 Glitter Ave.               Miami, FL
            USA                     Student
Jim Smith                     2341 West Hauser Street        Anytown, Anywhere
            BANZAI          V2E 2W2 Buckaroo
Dave Wilson                   123 Harrison Street            Sumas, WA
            USA                     Student
IMS:
```

This mouthful is still a valid IMS statement. What it is saying here is that if the country field is "CANADA", and the postal code is "V2T-1E6" then the comment field becomes "My neighbour", and the name of the person is PRINTed out.

Another file command is DELETE. This file command will go through all the records, deleting those which meet the range specifications. So,

```
DELETE ALL PRINT name
```

is a file command statement that will delete all the records from the file, and print out the name field of each. Obviously, this is a command you would not normally want to give to IMS.

```
DELETE ALL FOR comment="a moocher" PRINT NAME;
     " is deleted"
```

This file command will go through the file and delete all the records that have a comment field of "a moocher", print each of their names with " is deleted" following their names.

```
Wimpy is deleted
Freeloader Fred is deleted
The Sponger is deleted
```

is an example of what might be printed out to the screen.

If you want the output from the screen to go to another device, like a printer or file, the following commands are used:

```
SET PRINT TO "pathlist"
SET PRINT ON
```

where **pathlist** is the name of a file or device. This sends the PRINT output to the **pathlist** device, but the SET PRINT ON statement must be used to activate this. Output will still go to the screen.

If your printer was device **/p.** Then you would type the following two lines:

```
SET PRINT TO "/p"
SET PRINT ON
```

to send what normally displays on the screen to the printer. Enter some commands while the output is being sent to the printer. This is an easy way to keep a record of what you are doing.

If you want to turn the output to the printer or file off, type:

```
SET PRINT OFF
```

Output will no longer go to the printer but the path to the printer or file will stay open, to close it type:

```
SET PRINT TO ""
```

You now know some of the things you can do in IMS interactive mode. Don't be afraid to try different range specifications.

Once you have finished experimenting, type END or press the ESC key to bring you back to the main menu.

```
NOTE This is the file creator for lesson two
FILE lesson2


FIELD TEXT name OF 30
FIELD TEXT street OF 40
FIELD TEXT city_province OF 40 ALIAS cp
FIELD TEXT country OF 20
FIELD TEXT postal_code OF 6 MASK "L#L #L#"
FIELD DATE startdate
FIELD DATE enddate
FIELD REAL giftvalue MASK "#####.##"
FIELD INTEGER satisfaction
FIELD INTEGER ndays

KEY TEXT name OF 30 = CAP$(name)
KEY REAL giftval = giftvalue
KEY INTEGER sat = satisfaction
```

LESSON 2

Creating Files, Forms, and Programs

Objectives:

- using the text editor
- creating files
- creating forms

From the IMS main menu choose option 1, the text editor. Answer the prompt with **lesson2.imo**, and press the RETURN or ENTER key.

Type in the statements on the opposite page. What you are typing is called a file descriptor; these statements create a file with the fields of a mailing list similar to, but better than, what we saw in LESSON 1. It is important to know that when writing anything in IMS, no distinction is made between upper and lower case letters. Blank lines and extra spaces between words are also ignored so you can type them in if you wish.

You are presently using the text editor, a general purpose and easy to use program for typing and editing any sort of text. For the full story on the text editor, check the appendix. The editor has all the usual block editing capabilities, as well as search and replace abilities. Without repeating the manual, suffice it to say the the cursor keys as well as the DEL key have their usual meaning, HOME and ^E for END do just that, and ESC will move you between the text screen and the command menu. Help is always available by pressing ESC and then **?**. The menus on the status line will help you, but if you get into trouble check the manual or ask for a HELP screen. One more point, ^A, (control A), for Abort, will stop an action from occurring (such as a FIND operation).

An explanation of the statements follows below:

NOTE This is the file creator for lesson two

Any statement preceded by NOTE is a comment to the person reading the statement. A NOTE statement does not do anything to IMS, it is simply a way of telling the reader what the following statements are supposed to do.

FILE lesson2

```
NOTE This is the file creator for lesson two
FILE lesson2


FIELD TEXT name OF 30
FIELD TEXT street OF 40
FIELD TEXT city_province OF 40 ALIAS cp
FIELD TEXT country OF 20
FIELD TEXT postal_code OF 6 MASK "L#L #L#"
FIELD DATE startdate
FIELD DATE enddate
FIELD REAL giftvalue MASK "#####.##"
FIELD INTEGER satisfaction
FIELD INTEGER ndays

KEY TEXT name OF 30 = CAP$(name)
KEY REAL giftval = giftvalue
KEY INTEGER sat = satisfaction
```

This is a FILE statement, and it should be near the top. It tells IMS that the data files we will be creating will be called **lesson2.ida** and **lesson2.iin** in the disk directory. File names must start with a letter and then may have digits and underline characters ("_"). The maximum number of characters in a file name is 29. Because 4 extra characters are added to a file name, the name following FILE can be no longer than 25 characters. So "employee_data", "customer_accounts", and "magazines_cross_ref" are all valid file names. "3rd_version AR" is not valid (starts with a number), and "total$" is also not valid (it has the dollar sign).

FIELD TEXT name OF 30

This statement says that there is a field in the file called "name". It is a TEXT field, meaning that this field holds words, characters, or names ,ie. text, but it is not a number field. You could not add or subtract with this field. This makes sense since it is the name of a person, and you wouldn't want to add or subtract a name. It is also "OF 30", meaning that this field has a maximum length of 30 characters. Field names are restricted in a way similar to file names; they must start with a letter and then must contain letters, digits, or the underline character. The big difference is that they don't have a size limit; they can be any number of characters (within reason!).

FIELD TEXT street OF 40

This is another TEXT field. This one is called "street", has a maximum length of 40 characters, and is for the street address of the person.

FIELD TEXT city_province OF 40 ALIAS cp

This is another TEXT field, called "city_province" and it has a maximum length of 40 characters as well. It is for the city and province (or state) address of a person. The "ALIAS cp" part means that this field has two names, "city_province" and "cp". This comes in handy because the full name spells out the purpose of the field, and the alias becomes the abbreviation for the field.

FIELD TEXT country OF 20

This is a TEXT field, called "country", with a maximum length of 20 characters to hold the name of the country.

FIELD TEXT postal_code OF 6 MASK "L#L #L#"

This TEXT field has a maximum length of 6 characters and is for the postal code, of the person. This field also has a "mask" which specifies how this field must be entered and displayed. This mask specifies that 6 characters are needed, a letter (converted to upper case by L), then a number (described by a "#" symbol, then a capital letter, then

```
NOTE This is the file creator for lesson two
FILE lesson2


FIELD TEXT name OF 30
FIELD TEXT street OF 40
FIELD TEXT city_province OF 40 ALIAS cp
FIELD TEXT country OF 20
FIELD TEXT postal_code OF 6 MASK "L#L #L#"
FIELD DATE startdate
FIELD DATE enddate
FIELD REAL giftvalue MASK "#####.##"
FIELD INTEGER satisfaction
FIELD INTEGER ndays

KEY TEXT name OF 30 = CAP$(name)
KEY REAL giftval = giftvalue
KEY INTEGER sat = satisfaction
```

a space will be displayed in the field, then a number, letter, number sequence. Examples of this Canadian style postal code are "V3R 3E4" and "W7B 5F2". See the full details about masks under "MASK" in the reference section.

FIELD DATE startdate

This is a DATE field, meaning that the contents of the field can only be thought of as a date. This field is to contain the date the person first started sending you gifts or cards for Christmas. Dates are entered according to the default date format, which is month day year. The month can be the month name, or a three character abbreviation, or the month number. The year can be all four digits, like "1985", or just the last two, "85". If you want the date entered in a different format, use the "SET DATE TO" statement as noted in the reference manual under SET.

FIELD DATE enddate

This is another DATE field, meaning the contents of the field can only be thought of as a date. This field is to contain the last date on which you received a gift or cards from this person.

FIELD REAL giftvalue MASK "#####.##"

This is a REAL field. A REAL is a number that can have decimal places. REALs are numbers like 2.345, or -3456.1, or even 2.34E+10 (which is scientific notation for $2.34 \times 10^{+10}$, or 23400000000). In this case the field, called giftvalue, is to store a dollar and cents amount of the last gift sent to you by this person. A mask is also specified, showing that this field will have 2 decimal places and up to 5 digits in front of the decimal point.

FIELD INTEGER satisfaction

This is an INTEGER field. An integer is also a number but it has no decimal in it. 234 and -634 are integers, but 312. and 342.45 are REALs and not integers. This field is to hold a number between 0 and 100 indicating the level of satisfaction you got out of the last gift from this person.

FIELD INTEGER ndays

This is another INTEGER field. Its purpose is to tell the number of days before Christmas you should mail your gift to this person.

KEY TEXT name OF 30 = CAP$(name)

This is a key, not a field. Keys are how the files can be ordered when you process them. This statement is to set up a TEXT key called **name** with a maximum length of 30 characters. This key is to be on the name field after it has been converted to capital letters (CAP$). In other words, this key allows us to search, print out, and work

```
NOTE This is the file creator for lesson two
FILE lesson2


FIELD TEXT name OF 30
FIELD TEXT street OF 40
FIELD TEXT city_province OF 40 ALIAS cp
FIELD TEXT country OF 20
FIELD TEXT postal_code OF 6 MASK "L#L #L#"
FIELD DATE startdate
FIELD DATE enddate
FIELD REAL giftvalue MASK "#####.##"
FIELD INTEGER satisfaction
FIELD INTEGER ndays

KEY TEXT name OF 30 = CAP$(name)
KEY REAL giftval = giftvalue
KEY INTEGER sat = satisfaction
```

with the file in the order of the name field. CAP$ means that we will order the file according to the upper case version of the field. This is necessary because if it were not there the ordering would be by ASCII (see appendix for the ascii ordering table), which makes lower case letters come after upper case letters. For example, "smith" would come after "Yonnie".

KEY REAL giftval = giftvalue

This is another key, called **giftval**, of type REAL. It is of type REAL because the key is in order of the field **giftvalue** that we saw before, which is also a REAL field. Keys should always be of the same type as the field they are based on.

KEY INTEGER sat = satisfaction

This is the last key, called "sat". This key is of type INTEGER, because it is based on the field "satisfaction". This key will allow searches, print outs, etc. to be done in the order of how much satisfaction came from the last present you got from each person.

You should notice the order that the fields and keys are in. A NOTE statement comes first, telling the purpose of the file you are creating. NOTEs can actually come anywhere in the module to explain what a field does or what a key does. The FILE statement should come next, then the fields, and finally the keys. Keys can only be based on fields that exist in the same file you are creating. Also, remember to keep the number of keys low so that file performance remains acceptable. In this example we have three different keys, actually one or two keys are best.

Also notice the order of the words in the IMS statement. The first word on a line is always NOTE, FILE, HEADER, FIELD or KEY. The words HEADER, FIELD or KEY ar followed by the data type TEXT, REAL, INTEGER, LONG, or DATE. The name of the field or key comes next, and if it is a TEXT field the OF **length** follows. If it is a KEY the = **field expression** comes next. For a HEADER or FIELD, a MASK may be specified to control the format of the data. Last of all is an optional ALIAS **field name** part.

Save the text you have created by pressing **ESC** and then **S**. Answer the prompt with a carriage return; note that the filename you stated when you started the text editor is used when you save the text. Having saved the text, type **ESC** and then **Q** to quit the editor and go back to the main menu.

ULTRA MAIL LIST AND GIFT REMINDER
==========

NAME:
STREET:
CITY AND PROVINCE:
COUNTRY:
POSTAL CODE:
INITIAL GIFT DATE:
LAST GIFT DATE:
VALUE OF PRESENT:
LEVEL OF SATISFACTION (1-100):
DAYS BEFORE CHRISTMAS:

Row: 16 Column: 39

From the main menu choose number 2, to generate a data file from this file descriptor. This will prompt:

Name of file descriptor:

Answer the prompt with **lesson2.ide**, the same name used before when we entered the text editor. If any errors are reported during the generation of the data file, check your file for spelling mistakes.

Now that we have the data file we can create a form to maintain the file. Basically the procedure to create a form is to name the file you are making a form for and then "painting" the screen with the prompts and titles you think should go on the form. You also place the fields on the screen in the location you want them displayed.

When you are finished you tell IMS to generate a module to edit the file using this form. This module is complete enough so that you can compile and execute it from the main menu to add, delete, edit, and find records from the file. Or you can use the text editor to add capabilities to the module. In this lesson we are going to create the form and then examine the module IMS produces.

From the main menu select **3** to paint the form on the opposite page. Respond with **lesson2** when asked for the data file(s) to use. Simply move the cursor around the screen and produce a copy of what you see on the opposite page. When you are finished, move your cursor to a couple of spaces past NAME:. Press **^F** for field editing and then press **A** for adding a field. This will show a screen like:

```
FILE lesson2 FIELD TEXT name OF LENGTH 30
FILE lesson2 FIELD TEXT street OF LENGTH 40
FILE lesson2 FIELD TEXT city_province OF LENGTH 40 ALIAS cp
FILE lesson2 FIELD TEXT country OF LENGTH 20
FILE lesson2 FIELD TEXT postal_code OF LENGTH 6
FILE lesson2 FIELD DATE startdate
FILE lesson2 FIELD DATE enddate
FILE lesson2 FIELD REAL giftvalue
FILE lesson2 FIELD INTEGER satisfaction
FILE lesson2 FIELD INTEGER ndays
```

A **^F A** combination will show a screen with the fields of the files chosen when we started the forms editor. We chose file **lesson2** when we started the forms editor, and now the screen shows the fields from that file. Now, by pressing the cursor keys we can move the cursor over different fields. Move the cursor onto the line with the **name** field and press the RETURN or ENTER key. The screen will change and show our form again, with a series of "*" characters

ULTRA MAIL LIST AND GIFT REMINDER
==========

```
                    NAME: ******************************
                  STREET: ****************************************
          CITY AND PROVINCE: ****************************************
                 COUNTRY: ********************
             POSTAL CODE: L#L #L#
        INITIAL GIFT DATE: ******************
           LAST GIFT DATE: ******************
         VALUE OF PRESENT: #####.##
LEVEL OF SATISFACTION (1-100): ######
    DAYS BEFORE CHRISTMAS: ######
```

Row: 1 Column: 1

indicating the mask of the field. (The complete story on MASKs is in the reference manual under MASK.)

What we have done is tell IMS that we want to ENTER and DISPLAY the name field from the file lesson2 at this screen location. Move the cursor down one line to right after STREET ADDRESS: on the form and press **^F A**. This brings us back to our display of fields again, but now we see that the name field line is hi-lited. This shows us that the name field has already been chosen and can't be chosen again, unless it is "unchosen". Move the cursor to the street field line and press RETURN or ENTER. We are now back in the form and we can see the default mask for the street field. Continue in this way until all of the fields have been chosen. You should get a screen looking like the one on the opposite page.

As you are typing you may want to do some editing of the form. The **DEL** key will delete a character to the left and move the right part of the line over one character. **^D** (control D) is similar but it deletes the character under the cursor. **^C** will insert a space into the present line. **^I** will insert a blank line onto the screen, losing the bottom line. **^X** key sequence will delete the present line from the screen and bring in a new blank line at the bottom of the screen. Note that deleting a line is never possible when the line to be deleted has a field on it; the field must first be deleted (by **^F D**). The **HOME** and **^E** key sequences move to start and end of the screen, respectively. **^A** is a special key sequence that aborts any editing operation.

If you ever want to "unchoose" a field, move the cursor onto the mask of the field, and press **^F D**. This will take the mask off the screen, and take the field off the chosen list. Then the field can be chosen again for some other place on the screen.

When you are finished "painting" the form, press **ESC S**, for SAVE. This will give the question:

```
SAVE; filename:
```

Here IMS asks for the file name to save the form. Type in **lesson2** and press the RETURN or ENTER key.

```
                ULTRA MAIL LIST AND GIFT REMINDER
                =================================

                         NAME: ******************************
                       STREET: ****************************************
              CITY AND PROVINCE: ****************************************
                      COUNTRY: ********************
                  POSTAL CODE: L#L #L#
             INITIAL GIFT DATE: ******************
                LAST GIFT DATE: ******************
              VALUE OF PRESENT: #####.##
   LEVEL OF SATISFACTION (1-100): ######
          DAYS BEFORE CHRISTMAS: ######



GENERATE; name of output file:
```

When the file has been saved, press **ESC G** to generate an IMS program to help you maintain the information in the file. You will be prompted:

GENERATE; name of ouput file:

Type **lesson2.** Next it will prompt:

Form to use:

Type in the same name as you gave when you SAVEd the form (**lesson2**). A program will now be generated to help you manage your data.

Other important **ESC** and **^F** sequences are:

ESC Q for quit will give the question:

QUIT; Are you sure? (Y/N):

Pressing **Y** will quit the form and forget all changes to the form since you last SAVEd it. Pressing **N** will go back to the form.

ESC L for load, will give the prompt:

LOAD; filename:

Typing in a file name will cause IMS to look for a forms file with that file name, and load that information into the form destroying what was previously in the form.

ESC P for pass, will give the prompt:

PASS: command to operating system:

You can now type in any operating system command, for example **dir.**

ESC C for clear, clears the form on screen and all fields chosen before become "unchosen".

ESC O for output hardcopy, puts on the device of your choice a report of your form.

ESC ? or **ESC H** will give a help screen telling the important facts about the forms editor.

ULTRA MAIL LIST AND GIFT REMINDER
==========

NAME:
STREET:
CITY AND PROVINCE:
COUNTRY:
POSTAL CODE:
INITIAL GIFT DATE:
LAST GIFT DATE:
VALUE OF PRESENT:
LEVEL OF SATISFACTION (1-100):
DAYS BEFORE CHRISTMAS:

FILE: lesson2 KEY: NOKEY FORM: lesson2 RECORD #0

^F M for mask, lets you change the mask of a field that is already displayed on screen.

^F I for info, tells you which field the cursor is on.

^F D for delete, deletes the field from the screen allowing you to select it at a different location on the screen.

After saving the form, and generating a module quit the forms editor with **ESC Q**. From the main menu, choose option 5 to compile the IMS module generated by the forms editor, and answer the prompt with **lesson2**. After it has finished, choose option 6 to execute the module, and answer the prompt with module name **lesson2**. Notice that it acts very much like what we saw in lesson 1. You can add, delete, edit, and find records in the same manner. In fact, lesson 1 is a module that was generated by exactly the same process as was lesson 2.

Type data into this form. Practice inserting, deleting, modifying, searching, and adding records. Try changing the current key. When you have entered about a dozen records, quit to the main menu. Enter the interactive environment (option 7 from the main menu) and experiment as you did in lesson 1. For example:

```
LIST KEY giftval PRINT name
```

will PRINT out the names of the people in your list in order of the value of the gift they gave to you.

```
LIST KEY sat PRINT name
```

will do a similar thing but the names will be printed out in the order of your satisfaction with the gift they gave.

The point is that since our data file has more than one key we can specify what key we want by putting in the KEY **key name** clause. One kind of key is special and is present in every file, and it is called NOKEY.

```
LIST NOKEY PRINT name
```

This specification would PRINT out the names of the people on your mailing list in an order similar to the order you typed them in.

When you are finished, go back to the main menu by pressing the ESC key or typing END.

HEADER;

REPORT UPDATE FOR ULTRA MAIL LIST AND GIFT REMINDER

```
================================================================================
Name and Address                                    Gift Value    Satisfaction
--------------------------------------------------------------------------------
```

FOOTER;

```
================================================================================
                                    PAGE
```

PRIMARY FILE;

PRIMARY SUBTOTALS;
TOTALS;

```
--------------------------------------------------------------------------------
Total Number of People:                 Total Gift value:
```

END OF REPORT.

Line: 1 Row: 1 Column: 1

LESSON 3

REPORTS

Objectives:

- to learn how to use the reports form editor
- to learn how to generate and execute a report module

In this lesson, we will use the reports form editor to generate a program which will perform a report on a data base. We will build on the example given in lesson 2. From the main menu, enter selection 4, to describe a report form. When the prompt

Data base file(s):

appears, type **lesson2** and press the RETURN or ENTER key. This will bring up the reports form editor, presenting you with a blank report form. At this point you may start "painting" the report form presented on the opposite page.

Using the report writer is very similar to using the forms editor. Cursor movement, and line and character insertion and deletion are identical to the equivalent functions in the forms editor; so are functions which are initiated by pressing the **ESC** key. The **^B** key initiates a menu from which you may change the left or bottom borders of the report by pressing **L** or **B**, respectively. The **^F** key initiates a function menu, the contents of which depend on what line your cursor is on in the report form. The most common state of the **^F** menu will be identical in function and appearance to the **^F** menu of the forms editor - field editing.

A report form consists of a number of well defined sections. Each section has a title, usually displayed in inverse or half intensity video. After any section title, the user may insert new text lines by simply typing **^I**, as in the forms editor. Once a text line has been inserted, text and fields may be placed anywhere on that line, in the location in which they are to be displayed.

In order to place a field on the report form, move the cursor to the place (in a previously inserted text line) you wish the field to be. At this point, type **^F**. Instead of the Add option available in the forms editor, you will need

[redacted] HEADER; [redacted]

REPORT UPDATE FOR ULTRA MAIL LIST AND GIFT REMINDER

```
================================================================================
Name and Address                                    Gift Value    Satisfaction
--------------------------------------------------------------------------------
```

[redacted] FOOTER; [redacted]

```
================================================================================
                                  PAGE ###
```

[redacted] PRIMARY FILE; [redacted]

```
******************************                    #####.##      ######
****************************************
****************************************
******************** L#L #L#
```

[redacted] PRIMARY SUBTOTALS; [redacted]
[redacted] TOTALS; [redacted]

```
--------------------------------------------------------------------------------
Total Number of People: ####       Total Gift value: #####.##
```

[redacted] END OF REPORT. [redacted]

Line: 1 Row: 1 Column: 1

to choose from one of Print, Sum, Number, or Today. Note also that not all these field displaying options are available all the times.

The Print option is essentially identical to the Add option of the forms editor; when a program is run which was generated by the report writer, the data currently in that field will be printed out. The Sum option is nearly the same as Print, except that instead of printing out the current value of that field, the generated program will print out the sum of the data field as that portion of code is iterated (repeated). This may in fact take the form of a total, subtotal, or a running sum. Finally, the Number and Today options operate similarly to the Print option, except that instead of printing a field, they print the current page number or current date, respectively.

Of course, the Mask, Delete, and Info options available with the field editing command are the same as described in the forms editor.

So, proceed then with painting the form given opposite. Remember that to enter the fields and text in the HEADER, FOOTER, PRIMARY FILE, and TOTALS sections, you must have inserted blank lines after each respective title with **^I**.

The mask in the FOOTER section is the page number, selected by typing **^F N**. In the PRIMARY FILE section select the appropriate fields with the **^F P** function. Finally, in the TOTALS section, select both fields shown in that section with the **^F S** function; note that the **####** mask is really the **lesson2.name** field, but must be remasked after it is selected. You must also set the bottom margin to a value of 62, by pressing **^B B** at any point and typing the number 62.

When you are finished painting the report form, type **ESC S** for save. This will give the prompt:

```
SAVE; file name:
```

Type **lesson2** in response, and press RETURN. After the report form has been saved, type **ESC G** to generate a report module. The computer will ask:

```
GENERATE; name of output file:
```

Answer **lesson2rep**; this file will contain the IMS source file for the report module. When the computer prompts:

```
Index the primary data base (Y*/N) ?
```

type **N** in response.

[redacted] HEADER; [redacted]

REPORT UPDATE FOR ULTRA MAIL LIST AND GIFT REMINDER

```
================================================================================
Name and Address                                    Gift Value    Satisfaction
--------------------------------------------------------------------------------
```

[redacted] FOOTER; [redacted]

```
================================================================================
                                  PAGE ###
```

[redacted] PRIMARY FILE; [redacted]

```
*****************************                  #####.##       ######
*****************************************
*****************************************
******************** L#L #L#
```

[redacted] PRIMARY SUBTOTALS;
TOTALS; [redacted]

```
--------------------------------------------------------------------------------
Total Number of People: ####        Total Gift value: #####.##
```

[redacted] END OF REPORT. [redacted]

Line: 1 Row: 1 Column: 1

When you have finished, quit the reports form editor by typing **ESC Q**. Once you are back in the main menu, compile the source for the report module (selection 5), and execute it (selection 6). When the report module is executed, it will first ask you:

Output device to use:

Here you must enter the device you want the report printed on. Normally, you would type **/pl** or **/p** in response. After you have answered this question, you will be asked:

Single sheets or Continuous paper (S/C):

Enter either **S** or **C** for the style of paper feed you want.

After these questions, the report will start printing. At any time, you may type **ESC** in order to interupt printing. When you do so, you will be asked (at your terminal):

Report interupted; Abort or Continue (A/C)?

at which time you may press **A** to terminate the report or **C** to resume. Also, if an error occurs any time during execution, the message for that error will be printed, and you will be asked:

Abort, Ignore, or Retry (A/I/R)?

Abort will terminate the report, Ignore will simply resume execution, and Retry will attempt to re-execute the statement that caused the error to occur.

Since the report module was in fact generated in the IMS applications language, you can alter the source to any special requirements you may have for your particular report. The generator produces fairly modular code, so going through it will be easy; it is not recommended that you do so, however, until you more fully understand the IMS applications language. In lesson 4D, you will see a more complex application for the reports form editor, where there are two data bases.

LESSON 4

PAYROLL

Objectives:

- to create a payroll program

In this section we will create a payroll program. This payroll program is only an example and may not be applicable in a real setting, rather it is to teach the more involved aspects of IMS.

The steps involved in the payroll cycle are:

1. Data will be entered into the employee information file and into the job_data file.
2. Then hours are entered into the hours worked file.
3. After this a posting module reads through the hours worked file. It uses the pay rate and deduction rate information in the employee_data file to calculate the gross wages and deductions. It then updates the employee record and job record totals, and generates a series of records in the check_data file for wages to be paid to the employees.

This payroll programs will maintain several data files:

1. Employee data.
2. Job data.
3. Hours worked data.
4. Payroll checks.

One last module serves as a menu; it prints out the list of operations the payroll does, and then gets the response and executes the module to do that operation.

Writing this payroll program in a standard language would be a formidable challenge; we will now see how IMS makes it much easier.

```
FILE employee_data

HEADER INTEGER total_employees ALIAS tot_emp
HEADER REAL total_pay MASK "##########.##"
HEADER REAL total_deductions(4) MASK "##########.##" ALIAS tot_ded

FIELD TEXT name OF 25
FIELD TEXT address1 OF 30
FIELD TEXT address2 OF 30
FIELD TEXT address3 OF 30
FIELD INTEGER employee_no ALIAS emp_no
FIELD TEXT phone_no OF 7 MASK "###-####"
FIELD DATE birthdate
FIELD REAL deduction_rate(4) MASK "##.##" ALIAS ded_rate
FIELD REAL gross_ytd MASK "######.##"
FIELD REAL deduction_ytd(4) MASK "######.##" ALIAS ded_ytd
FIELD REAL net_ytd MASK "######.##"
FIELD TEXT salaried OF 1 MASK "L"
FIELD REAL salary MASK "######.##"
FIELD REAL regular MASK "##.##"
FIELD REAL overtime MASK "##.##"

KEY TEXT name OF 30 = CAP$(name)
KEY INTEGER emp_no = employee_no
```

LESSON 4A

PAYROLL - Employee Maintenance

Objectives:

- to create an employee maintenance module
- to learn about HEADER records
- to learn about arrays

This section will create the module to maintain the information relating to the employees on the payroll. Enter the text editor and answer the prompt with **empdata.ide** (this is a descriptor file). Type in the file structure on the opposite page; an explanation of the fields follows below:

FILE employee_data
The data will be stored on disk in the file called employee_data.

HEADER INTEGER total_employees ALIAS tot_emp
This is a header field, meaning that the file has this field only once (not one for every record). Header fields are useful for global information; ie., information about the file that pertains to all of the records. This header field contains the total number of employees on the payroll.

HEADER REAL total_pay MASK "##########.##"
This header field contains the total amount of wages paid to all the employees. The mask specifies a monetary style to be shown on the form, with 2 decimal places and 10 places in front of the decimal place. See the reference section under MASK for more details about masks.

HEADER REAL total_deductions(4) MASK "##########.##" ALIAS tot_ded
This is an array of four header fields. An array simply means that there is more than one item included in the field name. In this case there are 4 total deductions fields, tot_ded(1), tot_ded(2), tot_ded(3), and tot_ded(4). Each of these fields contains the total amount of each deduction taken from all of the employees. The mask specifies a large monetary amount.

FIELD TEXT name OF 25
The employee's name.

```
FILE employee_data

HEADER INTEGER total_employees ALIAS tot_emp
HEADER REAL total_pay MASK "##########.##"
HEADER REAL total_deductions(4) MASK "##########.##" ALIAS tot_ded

FIELD TEXT name OF 25
FIELD TEXT address1 OF 30
FIELD TEXT address2 OF 30
FIELD TEXT address3 OF 30
FIELD INTEGER employee_no ALIAS emp_no
FIELD TEXT phone_no OF 7 MASK "###-####"
FIELD DATE birthdate
FIELD REAL deduction_rate(4) MASK "##.##" ALIAS ded_rate
FIELD REAL gross_ytd MASK "######.##"
FIELD REAL deduction_ytd(4) MASK "######.##" ALIAS ded_ytd
FIELD REAL net_ytd MASK "######.##"
FIELD TEXT salaried OF 1 MASK "L"
FIELD REAL salary MASK "######.##"
FIELD REAL regular MASK "##.##"
FIELD REAL overtime MASK "##.##"

KEY TEXT name OF 30 = CAP$(name)
KEY INTEGER emp_no = employee_no
```

FIELD TEXT address1 OF 30
FIELD TEXT address2 OF 30
FIELD TEXT address3 OF 30
The employee's address as expressed on three lines. For example:

```
Jim Bodwin
23221 West Hauser St.
Anytown, Anywhere.
Anycountry
```

FIELD INTEGER employee_no ALIAS emp_no
The employee's number.

FIELD TEXT phone_no OF 7 MASK "###-####"
The employee's phone number. The mask specifies 3 digits, then a dash, then 4 digits format for the phone number.

FIELD DATE birthdate
The employee's birthdate.

FIELD REAL deduction_rate(4) MASK "##.##" ALIAS ded_rate
In this payroll a maximum of 4 deductions is allowed. Each of the deductions are expressed as a percentage of the gross in one of these 4 deduction rate fields. The mask specifies 2 digits in front of the decimal and two decimal places.

FIELD REAL gross_ytd MASK "######.##"
The employee's year to date gross pay. The mask specifies a monetary amount.

FIELD REAL deduction_ytd(4) MASK "######.##" ALIAS ded_ytd
The employee's year to date totals for 4 deductions.

FIELD REAL net_ytd MASK "######.##"
The employee's year to date net pay.

FIELD TEXT salaried OF 1 MASK "L"
This is a field that contains either a Y or N for yes or no. The purpose is to state whether the employee is salaried or not. Here the mask specifies one letter in the field, converted to upper case.

FIELD REAL salary MASK "######.##"
If the employee is salaried then this is the amount to be paid with each pay check.

FIELD REAL regular MASK "##.##"
If the employee is not salaried then this field is the regular hourly wage. Here the mask specifies 2 digits

```
FILE employee_data

HEADER INTEGER total_employees ALIAS tot_emp
HEADER REAL total_pay MASK "##########.##"
HEADER REAL total_deductions(4) MASK "##########.##" ALIAS tot_ded

FIELD TEXT name OF 25
FIELD TEXT address1 OF 30
FIELD TEXT address2 OF 30
FIELD TEXT address3 OF 30
FIELD INTEGER employee_no ALIAS emp_no
FIELD TEXT phone_no OF 7 MASK "###-####"
FIELD DATE birthdate
FIELD REAL deduction_rate(4) MASK "##.##" ALIAS ded_rate
FIELD REAL gross_ytd MASK "######.##"
FIELD REAL deduction_ytd(4) MASK "######.##" ALIAS ded_ytd
FIELD REAL net_ytd MASK "######.##"
FIELD TEXT salaried OF 1 MASK "L"
FIELD REAL salary MASK "######.##"
FIELD REAL regular MASK "##.##"
FIELD REAL overtime MASK "##.##"

KEY TEXT name OF 30 = CAP$(name)
KEY INTEGER emp_no = employee_no
```

before the decimal and two decimal places. The largest permissible value would be 99.99.

FIELD REAL overtime MASK "##.##"
If the employee is not salaried then this field is the overtime hourly wage.

KEY TEXT name OF 30 = CAP$(NAME)
A key in order of the employee name, case ignored.

KEY INTEGER emp_no = employee_no
A key in order of the employee number.

```
                    EMPLOYEE MAINTENANCE
                    ====================
            NAME: ******************************
         ADDRESS: ******************************
                  ******************************
                  ******************************
EMPLOYEE NUMBER: ######
           PHONE: ###-####
      BIRTH DATE: ******************

DEDUCTION RATES:
     1) ##.##      2) ##.##      3) ##.##      4) ##.##

YEAR TO DATE TOTALS:
         GROSS: ######.##
  DEDUCTIONS:
     1) ######.##  2) ######.##  3) ######.##  4) ######.##
          NET: ######.##

SALARIED (Y/N): L
   SALARY: ######.##          REGULAR: ##.##    OVERTIME: ##.##

                          Row: 1   Column: 1
```

Now choose menu option 3 - the option to paint a screen. Answer the prompt with the form name **employee_data.** Paint the screen as shown on the opposite page. Note that the header fields are not part of the form - they are handled by the payroll program. Go ahead and paint the screen as you did in lesson 2, going through all the non-header fields and assigning each a screen location. Don't forget to type **ESC S** to save the form. Give it the name **employee_edit.**

After you have saved the form, generate a file maintenance module with **ESC G**, naming it **employee_edit.** When asked for the name of the form, answer with **employee_edit.** When you quit from the forms editor, be sure to compile the new module. But before you compile it, you will have to edit the module, finding and deleting all OPEN and CLOSE statements. The reason for these changes is that the payroll menu program OPENs all the necessary files. If one of the auxilliary modules did also, the interpreter would signal an error when it tried opening the same data base twice.

```
FILE job_data

FIELD TEXT name OF 30
FIELD TEXT description OF 50
FIELD TEXT job_no OF 6 MASK "LL####"
FIELD REAL total_cost MASK "##########.##"
FIELD DATE start_date
FIELD DATE end_date
FIELD DATE complete_date

KEY TEXT job_no OF 6 = job_no
KEY TEXT name OF 30 = CAP$(name)
```

LESSON 4B

PAYROLL - Job Maintenance

Objectives:

- to create a job maintenance module

This section will create the data file and module which keeps track of the various jobs the company is working on. Enter the text editor with the file name **job_data.ide.** Type in the statements on the opposite page and create the file and module.

FILE job_data
The name of the file that will have the job data.

FIELD TEXT NAME OF 30
The name of the job.

FIELD TEXT description OF 50
A description of the job.

FIELD TEXT job_no OF 6 MASK "LL####"
The number of the job. Here the mask specifies the format to be 2 letters and then 4 digits. See the reference section under "MASK" for more information about masks.

FIELD REAL total_cost MASK "##########.##"
The total cost of the job. As far as payroll is concerned, an employees' wages for working on a job is the only way this field will be changed. The mask specifies a large monetary field.

FIELD DATE start_date
The date work started on the job.

FIELD DATE end_date
The date work ended on the job.

FIELD DATE complete_date
The date work was supposed to be completed on the job. The above three fields are not a concern to payroll but are included for completeness.

KEY TEXT job_no OF 6 = CAP$(job_no)
A key in order of the job number.

JOB MAINTENANCE
==========

NAME OF JOB: **********
DESCRIPTION: **********
JOB NUMBER: LL####
TOTAL COST: ##########.##
DATE WORK STARTED: **********
END WORK DATE (projected): **********
DATE JOB COMPLETED: **********

Row: 1 Column: 1

KEY TEXT name OF 40 = CAP$(name)
A key in order of the job name.

Now choose from the main menu option 3 to paint the screen form. Simply copy the opposite page onto your screen and select the fields in front of the prompts, as you did in lesson 2 and lesson 4a. After saving the form and generating an editing module named **job_edit**, edit that module, finding and deleting all OPEN and CLOSE statements. After this, compile the module.

```
FILE hours_data

FIELD LONG record_no
FIELD DATE workdate
FIELD TEXT job_no OF 6 MASK "LL####"
FIELD INTEGER employee_no ALIAS emp_no
FIELD REAL regular_hours MASK "##.##" ALIAS reg
FIELD REAL overtime_hours MASK "##.##" ALIAS ot

KEY INTEGER emp_no = emp_no
KEY TEXT job_no = job_no
```

LESSON 4C

PAYROLL - Hours Maintenance

Objectives:

- to create a hours maintenance module
- to learn about the file commands:
  FIND, EOF
- to learn about the statements:
  LABEL
  IF ... ELSE ... ENDIF
  LOCATE
  PRINT
  GOTO

This section will create the data file and module to maintain the hours worked information. Enter the editor with the file name **hours_data.ide** and type in the statements on the opposite page. Then create the file and compile the module as you did in lesson 4a. An explanation follows:

FILE hours_data
The data about the hours will be in file "hours_data".

FIELD LONG record_no
The record number of the hours worked report. This is a LONG field to prevent overflow on an integer value (LONG goes up to 2 billion).

FIELD DATE workdate
The date the hours were worked.

FIELD TEXT job_no OF 6 MASK "LL####"
The number of the job work was done on. The mask is as specified for the job number in lesson 4b.

FIELD INTEGER employee_no ALIAS emp_no
The number of the employee doing the work.

FIELD REAL regular_hours MASK "##.##" ALIAS reg
The regular hours worked. This mask is the same as the mask for the total hours worked in the employee_data file.

FIELD REAL overtime_hours MASK "##.##" ALIAS ot
The overtime hours worked. This mask is the same as the corresponding one in the employee_data file.

```
                HOURS MAINTENANCE
                -----------------

  REPORT NUMBER: ###########
           DATE: ******************
     JOB NUMBER: LL####
EMPLOYEE NUMBER: ######
  REGULAR HOURS: ##.##
 OVERTIME HOURS: ##.##


                                    Row: 1    Column: 1
```

KEY INTEGER emp_no = emp_no

A key to allow processing of the file in the order of the employee number.

KEY TEXT job_no OF 6 = job_no

Another key, to allow processing of the file in the order of the job_no.

Generate the data file and then from the main menu choose option 3 - to create a form. Copy the opposite page onto your screen and select the fields at the appropriate spot in front of the prompts, as you did before in lesson 4a and in lesson 2. Don't forget to save the module.

```
NOTE ***************************************************************
NOTE Main Loop


MODULE hours_edit

INTEGER keynum,temp
TEXT k OF 1
TEXT keystroke OF 1
TEXT keynam

keynum=1
OPEN 'hours_data'
USE FILE hours_data NOKEY

SET TRAP TO errtrap

LABEL restart
SET FORM TO 'hours'
GOSUB printscreen

LOOP
  GOSUB showkey
  GOSUB enterall
  GOSUB getchoice
  GOSUB dochoice
ENDLOOP

END


NOTE ***************************************************************
NOTE Reprint all fields on the screen


LABEL printscreen

DISPLAY record_no
DISPLAY workdate
DISPLAY job_no
DISPLAY employee_no
DISPLAY regular_hours
DISPLAY overtime_hours

RETURN


NOTE ***************************************************************
NOTE Return single keystroke from menu


LABEL getchoice

LOCATE 24,1
PRINT 'SELECT:  Insert Update Clear Delete First Last Next Previous Key Search Quit';
REPEAT
  LOCATE 24,8
  keystroke=CAP$(GETKEY)
  IF keystroke<' ' THEN
    PRINT '.';
  ELSE
    PRINT keystroke;
  ENDIF
UNTIL (keystroke=chr$(27)) OR ('IUCDFLNPKSQ' CT keystroke)
LOCATE 24,1
CLEAR LINE

RETURN


NOTE ***************************************************************
NOTE Select the appropriate action


LABEL dochoice

CASE
  WHEN keystroke='I' THEN
    INSERT
  ENDWHEN

  WHEN keystroke='U' THEN
    UPDATE
    GOSUB printscreen
  ENDWHEN

  WHEN keystroke='C' THEN
    CLEAR FORM
  ENDWHEN

  WHEN keystroke='D' THEN
    DELETE CURRENT
    IF RECORDNO>0 THEN
      GOSUB printscreen
```

Generate a form editing module with a name of **hours_edit.** Then exit to the main menu and choose option 1 to edit the module just generated. The purpose here is to add some checks to the editing of this form. Specifically, the job and employee numbers should be checked for validity when they were entered into the form. If they are not valid then the operator should be told this and given an opportunity to re-enter the values.

Once in the editor move the cursor to the start of the **ENTER job_no** line and type in the following line:

```
LABEL enter_job_no
```

LABEL is an IMS statement that marks a particular position of a module. In this case, we are marking the "ENTER job_no" line by the label "enter_job_no". The reason for this will become apparent in a moment.

Move the cursor down one line and type in the following:

```
FIND FILE job_data KEY job_no EXACT hours_data.job_no
USE FILE hours_data
IF EOF (job_data) THEN
   LOCATE (8,40)
   PRINT "ILLEGAL JOB NUMBER;  RE-ENTER"
   GOTO enter_job_no
ELSE
   LOCATE (8,40)
   PRINT job_data.name+"           "
ENDIF
```

These lines do the checking for a valid job number.

```
FIND FILE job_data KEY job_no EXACT hours_data.job_no
```

This is an IMS statement that searches in the file **job_data** (seen in lesson 4b) by its key **job_no** for a record with the job number field equal to the number in **hours_data.job_no. hours_data.job_no** refers to the job_no field in the hours_data file. **hours_data.job_no** has a period in the middle, this means that the part before the period (hours_data) is the file, and the part after the period (job_no) is a field in that file. This is in fact the same field referred to in the ENTER job_no statement. So what

```
      ELSE
        CLEAR FORM
      ENDIF
    ENDWHEN

    WHEN keystroke='F' THEN
      FIND FIRST
      GOSUB printscreen
    ENDWHEN

    WHEN keystroke='L' THEN
      FIND LAST
      GOSUB printscreen
    ENDWHEN

    WHEN keystroke='N' THEN
      FIND NEXT
      IF RECORD=0 THEN
        CLEAR FORM
      ELSE
        GOSUB printscreen
      ENDIF
    ENDWHEN

    WHEN keystroke='P' THEN
      FIND PREVIOUS
      IF RECORD=0 THEN
        CLEAR FORM
      ELSE
        GOSUB printscreen
      ENDIF
    ENDWHEN

    WHEN keystroke='K' THEN
      GOSUB dokey
    ENDWHEN

    WHEN keystroke='S' THEN
      FIND APPROX
      IF RECORD<>0 THEN
        GOSUB printscreen
      ENDIF
    ENDWHEN

    WHEN keystroke='Q' THEN
      CLOSE ALL
      END
    ENDWHEN
  ENDCASE

  RETURN
  NOTE **********************************************************
  NOTE Enter data into all input fields


  LABEL enterall

  LOOP
    ENTER record_no
    IF ESCAPE = 27 THEN EXIT : ENDIF
    ENTER workdate
    IF ESCAPE = 27 THEN EXIT : ENDIF
    ENTER job_no
    IF ESCAPE = 27 THEN EXIT : ENDIF
    ENTER employee_no
    IF ESCAPE = 27 THEN EXIT : ENDIF
    ENTER regular_hours
    IF ESCAPE = 27 THEN EXIT : ENDIF
    ENTER overtime_hours
    IF ESCAPE = 27 THEN EXIT : ENDIF
  ENDLOOP

  RETURN

  NOTE **********************************************************
  NOTE Select ne key field


  LABEL dokey

  CLEAR SCREEN

  PRINT
  PRINT 'Choose one field:'
  PRINT
  temp=1 : GOSUB indent
  PRINT 'NOKEY'
  temp=2 : GOSUB indent
  PRINT 'emp_no'
  temp=3 : GOSUB indent
  PRINT 'job_no'
  PRINT
```

happens is that the user ENTERs a job number into the field, and then the module looks in the job_data file to see if that number exists.

```
USE FILE hours_data
```

Reselects hours_data to be the current file. This is important to avoid confusion in any following field references and file commands.

```
IF EOF(job_data) THEN
```

Once we have done a FIND statement we have to see if it actually found anything. That is what this statement does. EOF is an IMS function that returns one - meaning a record was not found (the End Of File was reached during the previous FIND), or zero - meaning a record was found. This statement is saying "If the ENTERed job number was not found then ...".

```
LOCATE (8,40)
PRINT "ILLEGAL JOB NUMBER;  RE-ENTER"
GOTO enter_job_no
```

This is what is done if it was not found. First the message "ILLEGAL JOB NUMBER; RE-ENTER" is PRINTed at screen coordinates 8,40. LOCATE places the cursor at the eighth row and the fortieth column. Then the GOTO statement causes execution to continue at the label called enter_job_no. This means that the module will go back to ENTERing the job number field, to let the user enter a legal number.

```
ELSE
   LOCATE (8,40)
   PRINT job_data.name + "          "
ENDIF
```

The ELSE statement is part of the IF structure. ELSE always refers to the opposite condition of the preceding IF statement. The IF statement was testing whether there was no matching record in the job_data file. If that was true then the statements between the IF and the ELSE are executed. Then execution continues after the ENDIF. An ELSE checks for when the IF statement was <u>not</u> true (when there is a matching record in the job_data file) and executes the statements between the ELSE and the ENDIF. The PRINT statement shows the value of the name field in the job_data file, (plus extra spaces at the end to print over what was on the screen before), so the user can see if the job number entered is the right one. Finally, the ENDIF marks the end of the IF structure.

```
REPEAT
  PRINT 'Selection? ';
  INPUT temp
UNTIL temp>=0 AND temp<=3

CASE
  WHEN temp=1 THEN
    USE NOKEY
  ENDWHEN

  WHEN temp=2 THEN
    USE KEY emp_no
  ENDWHEN

  WHEN temp=3 THEN
    USE KEY job_no
  ENDWHEN

ENDCASE

IF temp>0 THEN
  keynum=temp
ENDIF

SET FORM TO 'hours'
GOSUB printscreen

RETURN


LABEL indent

IF keynum=temp THEN
  PRINT ' * ';
ELSE
  PRINT '   ';
ENDIF
PRINT temp;' - ';

RETURN


NOTE ****************************************************************
NOTE Display info on bottom line


LABEL showkey

CASE
  WHEN keynum=1 THEN
    keynam='NOKEY'
  ENDWHEN

  WHEN keynum=2 THEN
    keynam='emp_no'
  ENDWHEN

  WHEN keynum=3 THEN
    keynam='job_no'
  ENDWHEN
ENDCASE

LOCATE 24,1
PRINT 'FILE: hours_data    KEY: ';keynam;'    FORM: hours    RECORD #';RECORD;

RETURN


NOTE ****************************************************************
NOTE General error handler


LABEL errtrap

CLEAR SCREEN

PRINT
PRINT
PRINT
HELP ERROR
IF ERROR=2 THEN
  END
ENDIF
PRINT
PRINT
PRINT 'Type any key to continue: ';
k=GETKEY

RESUME AT restart


NOTE ****************************************************************
NOTE
```

That does it for checking the entered job number. The other field to check is the employee number field. Move the cursor to the start of the **ENTER employee_no** line and type in the following line:

```
LABEL enter_emp_no
```

This marks the next line with the name of "enter_emp_no". Move the cursor down one line and type in the following:

```
FIND FILE employee_data KEY emp_no EXACT
                                    hours_data.emp_no
USE FILE hours_data
IF EOF(employee_data) THEN
   LOCATE (9,40)
   PRINT "ILLEGAL EMPLOYEE NUMBER; RE-ENTER"
   GOTO enter_emp_no
ELSE
   LOCATE (9,40)
   PRINT employee_data.name+"                "
ENDIF
```

This is the same basic idea as with the job number field. First the user types in the employee number. Then the module FINDs the employee record with that employee number. If the record does not exist, ie. the end of the file was reached, then a message reporting this to the user is printed out and the module goes back to re-enter the field. If the field did exist then the name of the employee with the number is PRINTed out.

You should double check this module to see if it is correctly typed in. Compare it to the listing on the opposite page and correct any differences. Remember that in IMS modules the text can be in upper or lower case, and blank lines and extra spaces between words (except in TEXT constants) are ignored.

Finally, don't forget to find and delete any OPEN or CLOSE statments. Once you are finished, save the changes, exit the text editor and choose option 5 to compile the module.

```
FILE check_data

HEADER LONG check_req_no

FIELD INTEGER employee_number ALIAS emp_no
FIELD REAL gross MASK "######.##"
FIELD REAL net MASK "######.##"
FIELD REAL deduction(4) MASK "######.##" ALIAS ded

KEY INTEGER emp_no = employee_number
```

LESSON 4D

PAYROLL - Check_Data File

Objectives:

- to create a file for the check data
- to learn how to modify a report program.

In this section the file to hold the check information will be made. The check information is the wages and deductions for each employee. Notice that a module to edit them is not needed; they will be automatically maintained by a payroll module using the data from the other files. Enter the text editor and answer the prompt with **check.ide**. Type in the statements on the opposite page and create the file, but do not create a form to edit this file.

FILE check_data
The file holding the check information will be called "check_data".

HEADER LONG check_reg_no
This header field contains the check register number of the next check to be printed out.

FIELD INTEGER employee_number ALIAS emp_no
The number of the employee.

FIELD REAL gross MASK "######.##"
The gross amount of wages for the pay period. This mask specifies a monetary amount.

FIELD REAL deduction(4) MASK "######.##" ALIAS ded
The amount of each deduction for the pay period.

FIELD REAL net MASK "######.##"
The net amount of this check.

KEY INTEGER emp_no= emp_no
A key in order of the employee number.

```
HEADER;

                        CHECK INFORMATION REPORT

================================================================================
Number  Employee Name                  Gross        Deductions        Net
--------------------------------------------------------------------------------
FOOTER;
================================================================================
                                  PAGE ###
PRIMARY FILE;
######  ************************   ######.##   ######.##        ######.##
                                              ######.##
                                              ######.##
                                              ######.##
--------------------------------------------------------------------------------
PRIMARY SUBTOTALS;
LINK  FILE employee data KEY emp no  TO  FILE check data
SUBTOTALS;
TOTALS;
                                  TOTAL GROSS  TOTAL DEDUCTIONS  TOTAL NET
                                  ######.##    ######.##         ######.##
END OF REPORT.
```

Line: 1 Row: 1 Column: 1

After generating the check_data data base with main menu option 2, select the report writer, option 4. When the following prompt appears:

Data base file(s):

type **check_data employee_data.** Paint the report form which appears on the opposite page.

First set the left margin to a value of 1 by pressing **^B L**, then entering the number 1. Set the bottom margin to a value of 64 with **^B B** and entering the number.

The names of some of the data fields on the opposite page are not very obvious. In the PRIMARY FILE section, the field under the **Gross** title is check_data.gross. The four fields under the **Deductions** title are the four field array elements of check_data.ded: check_data.ded(1) through check_data.ded(4), and the field under the **Net** title is check_data.net. Place the above fields on the report form with the **^F P** command, and selecting the appropriate field. Note that the **Gross**, **Deductions**, and **Net** titles are actually entered in the HEADER section.

In the TOTALS section, the field under the **TOTAL GROSS** title is check_data.gross, the field under the **TOTAL DEDUCTIONS** title is check_data.ded(1), and the field under the **TOTAL NET** title is check_data.gross. These fields are summated, so they must be placed on the report form by typing **^F S** and selecting the appropriate field.

Once you have painted the report form, save it with the **ESC S** command sequence, answering the

SAVE; file name:

prompt with **check.** Once this is completed, type **ESC G** to generate the report program. Answer the

GENERATE; name of output file:

prompt with **check_report.** When the query

Index the primary data base (Y*/N) ?

appears, answer **Y**, then select the emp_no key for the check_data data base, and finally, press **ENTER** or **RETURN** in response to the following prompt

Field expression to match key:

when it appears. This last sequence of three questions by the computer is designed to allow the user to index the primary data base in a flexible manner.

Now that you have saved and generated your report, type **ESC Q** to quit the reports editor. From the main menu, enter the text editor (selection 1), and enter **check_report** in response to the prompt. When the text editor comes up, you will be editing the generated report program.

Search for the following line:

```
a1 = a1 + check_data.gross
```

change this to

```
a1 = a1 + check_data.gross - check_data.ded(1) -
check_data.ded(2) - check_data.ded(3) - check_data.ded(4)
```

(Note that this is a single line). Next, search for the following line:

```
a3 = a3 + check_data.deduction(1)
```

change this line to be:

```
a3 = a3 + check_data.ded(1) + check_data.ded(2) +
check_data.ded(3) + check_data.ded(4)
```

(This represents a single line). These changes are needed because the reports form editor does not allow the user to enter field expressions when printing or summing fields; only single data fields are allowed.

Next, find the line with the statement:

```
LABEL clean_up
```

Immediately after it, enter the following line:

```
UNLINK  FILE employee_data
```

This restores the state of the file relations prior to the execution of the report module.

Finally, you must also find and delete any OPEN or CLOSE statments. Once you have finished these changes, don't forget to save them before exiting to the main menu and compiling the program.

```
NOTE this is the posting part of the payroll program

MODULE paypost

real amount,deduct(4),temp
integer old_employee,count

USE FILE employee_data KEY emp_no
USE FILE job_data KEY job_no
FIND FILE hours_data KEY emp_no FIRST
WHILE RECORD<>0 DO
  amount=0,deduct(1)=0,deduct(2)=0,deduct(3)=0,deduct(4)=0
  old_employee=hours_data.emp_no
  FIND FILE employee_data EXACT old_employee
  WHILE NOT(EOF(hours_data)) AND hours_data.emp_no=old_employee DO
    IF employee_data.salaried="Y" THEN
      amount=employee_data.salary
    ELSE
      amount=amount+hours_data.reg*employee_data.regular+hours_data.ot*employee_data.overtime
    ENDIF
    FIND FILE job_data EXACT hours_data.job_no
    IF RECORD THEN
      .total_cost=.total_cost+amount
      UPDATE
    ELSE
      PRINT "Job #";hours_data.job_no;" not found."
    ENDIF
    FIND FILE hours_data NEXT
  ENDWHILE
  employee_data.total_pay=employee_data.total_pay+amount
  count=1, temp=0
  WHILE count<=4 DO
    deduct(count)=employee_data.deduction_rate(count)/100*amount
    employee_data.tot_ded(count)=employee_data.tot_ded(count)+deduct(count)
    employee_data.ded_ytd(count)=employee_data.ded_ytd(count)+deduct(count)
    check_data.ded(count)=deduct(count)
    temp=temp+deduct(count)
    count=count+1
  ENDWHILE
  employee_data.gross_ytd=employee_data.gross_ytd+amount
  employee_data.net_ytd=employee_data.net_ytd+amount-temp
  IF EOF(employee_data) THEN
    PRINT "Employee #";old_employee;" not found."
  ELSE
    UPDATE FILE employee_data
  ENDIF
  check_data.emp_no=employee_data.emp_no
  check_data.gross=amount
  check_data.net=amount-temp
  INSERT FILE check_data
  USE FILE hours_data
ENDWHILE

CLEAR FILE hours_data
END
```

LESSON 4E

PAYROLL - Putting It All Together

Objectives:

- to create the posting module
- to create the menu module
- to learn file commands
  - LINK
  - INSERT
  - CLEAR
  - REINDEX
- to learn control commands
  - WHILE ... ENDWHILE
  - IF ... ELSE ... ENDIF
  - LABEL
  - GOTO
  - CALL
  - CASE ... WHEN ... ENDWHEN ... ENDCASE
- and to learn the INPUT and CLEAR SCREEN command

This section will now put all the parts of the payroll package together. There are two things that still need to be done. First of all, the posting module which outputs records for the check writer has to be written, and finally a module which prints a menu of choices to operate payroll has to be created.

Enter the text editor with the file name **paypost.** Type in the statements on the opposite page. When finished typing, save the text and compile the module. This is the posting module which reads in all the hours records and updates the employee records, the job records, and writes the amount of each to the check_data file. An explanation of each of the lines follows below:

NOTE this is the posting part of the payroll program
This is a comment saying what the module is supposed to do.

MODULE paypost
This names the module as "paypost".

```
NOTE this is the posting part of the payroll program

MODULE paypost

real amount,deduct(4),temp
integer old_employee,count

USE FILE employee_data KEY emp_no
USE FILE job_data KEY job_no
FIND FILE hours_data KEY emp_no FIRST
WHILE RECORD<>0 DO
  amount=0,deduct(1)=0,deduct(2)=0,deduct(3)=0,deduct(4)=0
  old_employee=hours_data.emp_no
  FIND FILE employee_data EXACT old_employee
  WHILE NOT(EOF(hours_data)) AND hours_data.emp_no=old_employee DO
    IF employee_data.salaried="Y" THEN
      amount=employee_data.salary
    ELSE
      amount=amount+hours_data.reg*employee_data.regular+hours_data.ot*employee_data.ove
    ENDIF
    FIND FILE job_data EXACT hours_data.job_no
    IF RECORD THEN
      .total_cost=.total_cost+amount
      UPDATE
    ELSE
      PRINT "Job #";hours_data.job_no;" not found."
    ENDIF
    FIND FILE hours_data NEXT
  ENDWHILE
  employee_data.total_pay=employee_data.total_pay+amount
  count=1, temp=0
  WHILE count<=4 DO
    deduct(count)=employee_data.deduction_rate(count)/100*amount
    employee_data.tot_ded(count)=employee_data.tot_ded(count)+deduct(count)
    employee_data.ded_ytd(count)=employee_data.ded_ytd(count)+deduct(count)
    check_data.ded(count)=deduct(count)
    temp=temp+deduct(count)
    count=count+1
  ENDWHILE
  employee_data.gross_ytd=employee_data.gross_ytd+amount
  employee_data.net_ytd=employee_data.net_ytd+amount-temp
  IF EOF(employee_data) THEN
    PRINT "Employee #";old_employee;" not found."
  ELSE
    UPDATE FILE employee_data
  ENDIF
  check_data.emp_no=employee_data.emp_no
  check_data.gross=amount
  check_data.net=amount-temp
  INSERT FILE check_data
  USE FILE hours_data
ENDWHILE

CLEAR FILE hours_data
END
```

REAL amount,deduct(4),temp

amount is a variable that acts as the gross the employee will get paid, **deduct(4)** is an array that acts as the total deductions amount, and **temp** is a temporary holder for the sum of the deductions.

INTEGER old_employee,count

old_employee is a variable to keep track of the current employee as the hours file is read, and **count** is a variable to help read through the deduction arrays.

USE FILE employee_data KEY emp_no
USE FILE job_data KEY job_no
FIND FILE hours_data KEY emp_no FIRST

Set **emp_no** to be the default key for employee file, **job_no** to be the default key for job file, and finally, find the first record in the hours file, indexing that file by the employee number.

WHILE RECORD<>0 DO

WHILE is a kind of program control statement. It is saying that while something is true keep executing the part between this WHILE and the following matching ENDWHILE. RECORD is a function that returns the number of the current record. If the record were not found or there were no more records to go through, then RECORD is equal to zero. So WHILE RECORD <>0 DO is saying that while there are more records to read in the hours_data file do the following...

<> is called a **relational**, because it relates a value on the left to a value on the right of it. Symbols or letters can be used for relationals; the following list explains the common relationals:

=	or **EQ**	equal to
>	or **GT**	greater than
<	or **LT**	less than
>=	or **GE**	greater than
<=	or **LE**	less than
<>	or **NE**	not equal to

and for relating text values:

BW	begins with
CT	contains
SL	sounds like

So the above line could be: WHILE RECORD NE 0 DO

amount=0,deduct(1)=0,deduct(2)=0,deduct(3)=0,deduct(4)=0
old_employee=hours_data.emp_no

Initialize the variables. Set **amount** and the **deduct**

```
NOTE this is the posting part of the payroll program

MODULE paypost

real amount,deduct(4),temp
integer old_employee,count

USE FILE employee_data KEY emp_no
USE FILE job_data KEY job_no
FIND FILE hours_data KEY emp_no FIRST
WHILE RECORD<>0 DO
  amount=0,deduct(1)=0,deduct(2)=0,deduct(3)=0,deduct(4)=0
  old_employee=hours_data.emp_no
  FIND FILE employee_data EXACT old_employee
  WHILE NOT(EOF(hours_data)) AND hours_data.emp_no=old_employee DO
    IF employee_data.salaried="Y" THEN
      amount=employee_data.salary
    ELSE
      amount=amount+hours_data.reg*employee_data.regular+hours_data.ot*employee_data.overtime
    ENDIF
    FIND FILE job_data EXACT hours_data.job_no
    IF RECORD THEN
      .total_cost=.total_cost+amount
      UPDATE
    ELSE
      PRINT "Job #";hours_data.job_no;" not found."
    ENDIF
    FIND FILE hours_data NEXT
  ENDWHILE
  employee_data.total_pay=employee_data.total_pay+amount
  count=1, temp=0
  WHILE count<=4 DO
    deduct(count)=employee_data.deduction_rate(count)/100*amount
    employee_data.tot_ded(count)=employee_data.tot_ded(count)+deduct(count)
    employee_data.ded_ytd(count)=employee_data.ded_ytd(count)+deduct(count)
    check_data.ded(count)=deduct(count)
    temp=temp+deduct(count)
    count=count+1
  ENDWHILE
  employee_data.gross_ytd=employee_data.gross_ytd+amount
  employee_data.net_ytd=employee_data.net_ytd+amount-temp
  IF EOF(employee_data) THEN
    PRINT "Employee #";old_employee;" not found."
  ELSE
    UPDATE FILE employee_data
  ENDIF
  check_data.emp_no=employee_data.emp_no
  check_data.gross=amount
  check_data.net=amount-temp
  INSERT FILE check_data
  USE FILE hours_data
ENDWHILE

CLEAR FILE hours_data
END
```

array to zero and **old_employee** to the value of the employee number field in the current record of the hours_data file.

FIND FILE employee_data EXACT old_employee
Find the entry in the employee file for the employee number given in the hours file.

WHILE NOT(EOF(hours_data)) AND hours_data.emp_no = old_employee DO
This WHILE loop is for reading the hours records of each employee. It is saying that WHILE there are more records to read and the present record is still on the same employee (hours_data.emp_no = old_employee) do the following.

```
IF employee_data.salaried ="Y" THEN
  amount=employee_data.salary
```
If the employee is salaried then the amount is the employee's salary.

```
ELSE
  amount=amount+hours_data.reg*employee_data.regular+
                hours_data.ot*employee_data.overtime
```
If the employee is not salaried then the amount is the number of hours in the hours_data file times the rate in the employee_data file, both regular and overtime. This is added to **amount** because the employee may have many records in the hours file and the gross from each must be added into the running total.

ENDIF
End the previous IF statement.

FIND FILE job_data EXACT hours_data.job_no
Find the job file entry corresponding to the hours file job being processed.

```
IF RECORD THEN
  .total_cost=.total_cost+amount
  UPDATE
```
If a job file entry was found (IF RECORD THEN) update the total cost of the job (in the job file). Note the use of **.total_cost**: the "." forces the use of the current file.

```
ELSE
  PRINT "Job #";hours_data.job_no;" not found."
ENDIF
```
Warn the operator when no corresponding job is found, and terminate the IF statement.

FIND FILE hours_data NEXT
The next record in the hours_data file is found.

```
NOTE this is the posting part of the payroll program

MODULE paypost

real amount,deduct(4),temp
integer old_employee,count

USE FILE employee_data KEY emp_no
USE FILE job_data KEY job_no
FIND FILE hours_data KEY emp_no FIRST
WHILE RECORD<>0 DO
  amount=0,deduct(1)=0,deduct(2)=0,deduct(3)=0,deduct(4)=0
  old_employee=hours_data.emp_no
  FIND FILE employee_data EXACT old_employee
  WHILE NOT(EOF(hours_data)) AND hours_data.emp_no=old_employee DO
    IF employee_data.salaried="Y" THEN
      amount=employee_data.salary
    ELSE
      amount=amount+hours_data.reg*employee_data.regular+hours_data.ot*employee_data.ove
    ENDIF
    FIND FILE job_data EXACT hours_data.job_no
    IF RECORD THEN
      .total_cost=.total_cost+amount
      UPDATE
    ELSE
      PRINT "Job #";hours_data.job_no;" not found."
    ENDIF
    FIND FILE hours_data NEXT
  ENDWHILE
  employee_data.total_pay=employee_data.total_pay+amount
  count=1, temp=0
  WHILE count<=4 DO
    deduct(count)=employee_data.deduction_rate(count)/100*amount
    employee_data.tot_ded(count)=employee_data.tot_ded(count)+deduct(count)
    employee_data.ded_ytd(count)=employee_data.ded_ytd(count)+deduct(count)
    check_data.ded(count)=deduct(count)
    temp=temp+deduct(count)
    count=count+1
  ENDWHILE
  employee_data.gross_ytd=employee_data.gross_ytd+amount
  employee_data.net_ytd=emyee_data.net_ytd+amount-temp
  IF EOF(employee_data) THEN
    PRINT "Employee #";old_employee;" not found."
  ELSE
    UPDATE FILE employee_data
  EF
  check_data.emp_no=employee_data.emp_no
  check_data.gross=amount
  check_data.net=amount-temp
  INSERT FILE check_data
  USE FILE hours_data
ENDLE

CLEAR FILE hours_data
END
```

ENDWHILE

End the previous WHILE statement. This means that when the previous WHILE loop is finished, (which is when we finished reading all the hours records for the employee), we go to the next line after this ENDWHILE.

employee_data.total_pay=employee_data.total_pay+amount

Update the total_pay header field in the employee_data file.

count=1,temp=0
WHILE count<=4 DO

Assign **count** a value of 1, and then set up a WHILE loop that executes 4 times with count initially equal to 1, then 2, then 3, then finally 4. Additionally, temp is initialized to 0.

deduct(count)=amount*employee_data.deduction_rate(count)/100

calculate the next deduction amount.

employee_data.tot_ded(count)=employee_data.tot_ded(count)+deduct(count)
employee_data.ded_ytd(count)=employee_data.ded_ytd(count)+deduct(count)
check_data.ded(count)=deduct(count)

Update the deduction fields. The deduct(count) array elements are added to the employee file header fields and to the employee record year to date fields. In the check file, the deduction amount is placed in the deduction array for that particular check payment.

temp=temp+deduct(count)
count=count+1

Update the deductions total (in temp), and increment count in order to process the next deduction.

ENDWHILE

Loop back to the WHILE count<=4 DO statement in order to see if more deductions need to be done. If there are none, continue processing at the following statement:

employee_data.gross_ytd=employee_data.gross_ytd+amount
employee_data.net_ytd=employee_data.net_ytd+amount-temp

Update the employee gross year to date field, and the employee net year to date field.

```
NOTE this is the posting part of the payroll program

MODULE paypost

real amount,deduct(4),temp
integer old_employee,count

USE FILE employee_data KEY emp_no
USE FILE job_data KEY job_no
FIND FILE hours_data KEY emp_no FIRST
WHILE RECORD<>0 DO
  amount=0,deduct(1)=0,deduct(2)=0,deduct(3)=0,deduct(4)=0
  old_employee=hours_data.emp_no
  FIND FILE employee_data EXACT old_employee
  WHILE NOT(EOF(hours_data)) AND hours_data.emp_no=old_employee DO
    IF employee_data.salaried="Y" THEN
      amount=employee_data.salary
    ELSE
      amount=amount+hours_data.reg*employee_data.regular+hours_data.ot*employee_data.over
    ENDIF
    FIND FILE job_data EXACT hours_data.job_no
    IF RECORD THEN
      .total_cost=.total_cost+amount
      UPDATE
    ELSE
      PRINT "Job #";hours_data.job_no;" not found."
    ENDIF
    FIND FILE hours_data NEXT
  ENDWHILE
  employee_data.total_pay=employee_data.total_pay+amount
  count=1, temp=0
  WHILE count<=4 DO
    deduct(count)=employee_data.deduction_rate(count)/100*amount
    employee_data.tot_ded(count)=employee_data.tot_ded(count)+deduct(count)
    employee_data.ded_ytd(count)=employee_data.ded_ytd(count)+deduct(count)
    check_data.ded(count)=deduct(count)
    temp=temp+deduct(count)
    count=count+1
  ENDWHILE
  employee_data.gross_ytd=employee_data.gross_ytd+amount
  employee_data.net_ytd=employee_data.net_ytd+amount-temp
  IF EOF(employee_data) THEN
    PRINT "Employee #";old_employee;" not found."
  ELSE
    UPDATE FILE employee_data
  ENDIF
  check_data.emp_no=employee_data.emp_no
  check_data.gross=amount
  check_data.net=amount-temp
  INSERT FILE check_data
  USE FILE hours_data
ENDWHILE

CLEAR FILE hours_data
END
```

```
IF EOF(employee_data) THEN
  PRINT "Employee #";old_employee;" not found."
ELSE
  UPDATE FILE employee_data
ENDIF
```
Update the employee file if the employee number is valid.

```
check_data.emp_no=employee_data.emp_no
check_data.gross=amount
check_data.net=amount-temp
```
Update all the fields of the check_data file record. The number comes from the employee_data file, and the gross and net amounts come from variables already calculated.

INSERT FILE check_data
Add the record just assigned to the check_data file.

USE FILE hours_data
Set the default file to the hours file. This loads the RECORD variable with the record number the hours file is on.

ENDWHILE
End the first WHILE. This means that when the condition in the first WHILE is no longer true, (ie. there are no more records to read in the hours_data file), go to the next statement after this ENDWHILE.

CLEAR FILE hours_data
Delete all the hours_data file items, since they have all been processed.

END
The end of the module.

Here is a general explanation of this module. The module reads through all the records in the hours file - that is the purpose of the first WHILE statement. The second WHILE statement goes through all the records for a single employee. In this second WHILE loop the module calculates the total gross that the employee will earn by adding up the hours in the individual hour records. These hours are multiplied by the rates in the individual employee's record to get the gross pay, and the deductions are calculated from this gross pay.

When the next record is for a different employee, the module updates the employee fields to account for the new wages and deductions. The data for the check file is then put into a check data record and INSERTed into the file. At this point all the payroll information about the employee has been read and updated. The module goes on to the next employee. The second WHILE loop will then handle this next employee, then the next, ..., until all the employees have been done.

```
NOTE this is the menu to operate the payroll program

MODULE payroll

TEXT choice OF 1

OPEN "employee_data"
OPEN "job_data"
OPEN "hours_data"
OPEN "check_data"

LOOP
  CLEAR SCREEN
  PRINT
  PRINT
  PRINT "PAYROLL MAINTENANCE";TAB(40);today;"  ";time
  PRINT
  PRINT TAB(10);"0. Quit"
  PRINT TAB(10);"1. Employee Maintainance"
  PRINT TAB(10);"2. Job Maintainance"
  PRINT TAB(10);"3. Time Maintainance"
  PRINT TAB(10);"4. Post Time Reports"
  PRINT TAB(10);"5. Report On Posted Records"
  PRINT
  PRINT "ENTER YOUR SELECTION: ";
  choice=GETKEY
  IF choice<" " THEN
    PRINT "."
  ELSE
    PRINT choice
  ENDIF
  IF choice<"0" OR choice>"5" THEN
    REDO
  ENDIF

  CASE
    WHEN choice="0" THEN
      CLOSE ALL
      QUIT
    ENDWHEN
    WHEN choice="1" THEN
      CALL employee_edit
    ENDWHEN
    WHEN choice="2" THEN
      CALL job_edit
    ENDWHEN
    WHEN choice="3" THEN
      CALL hours_edit
    ENDWHEN
    WHEN choice="4" THEN
      CALL paypost
    ENDWHEN
    WHEN choice="5" THEN
      CALL check_report
    ENDWHEN
  ENDCASE
ENDLOOP
```

On the opposite page you see the module for the menu which controls all the features of the payroll program. Enter the text editor with the file name **payroll**. Type in the module, and when finished compile it. An explanation of the statements follows below:

NOTE this is the menu to operate the payroll program
This is just a comment on what the module is supposed to do.

MODULE payroll
This identifies the module as "payroll".

TEXT choice OF 1
This is a variable to contain the user's selection from the menu.

```
OPEN FILE "employee_data"
OPEN FILE "job_data"
OPEN FILE "hours_data"
OPEN FILE "check_data"
```
This opens all the files of the payroll program.

LOOP
LOOP is a statement that marks the beginning of a group of statements which may be repeated more than once.

CLEAR SCREEN
This will clear the screen.

```
PRINT
PRINT
PRINT "PAYROLL MAINTENANCE";TAB(40);TODAY;"  ";TIME
PRINT
PRINT TAB(10);"0. Quit"
PRINT TAB(10);"1. Employee Maintenance"
PRINT TAB(10);"2. Job Maintenance"
PRINT TAB(10);"3. Time Maintenance"
PRINT TAB(10);"4. Post Time Reports"
PRINT TAB(10);"5. Report On Posted Records"
PRINT
```
This series of PRINT statements prints out the menu. TODAY is a function that returns the current date, and TIME is another function that returns the current hours, minutes, and seconds.

```
NOTE this is the menu to operate the payroll program

MODULE payroll

TEXT choice OF 1

OPEN "employee_data"
OPEN "job_data"
OPEN "hours_data"
OPEN "check_data"

LOOP
  CLEAR SCREEN
  PRINT
  PRINT
  PRINT "PAYROLL MAINTENANCE";TAB(40);today;"  ";time
  PRINT
  PRINT TAB(10);"0. Quit"
  PRINT TAB(10);"1. Employee Maintainance"
  PRINT TAB(10);"2. Job Maintainance"
  PRINT TAB(10);"3. Time Maintainance"
  PRINT TAB(10);"4. Post Time Reports"
  PRINT TAB(10);"5. Report On Posted Records"
  PRINT
  PRINT "ENTER YOUR SELECTION: ";
  choice=GETKEY
  IF choice<" " THEN
    PRINT "."
  ELSE
    PRINT choice
  ENDIF
  IF choice<"0" OR choice>"5" THEN
    REDO
  ENDIF

  CASE
    WHEN choice="0" THEN
      CLOSE ALL
      QUIT
    ENDWHEN
    WHEN choice="1" THEN
      CALL employee_edit
    ENDWHEN
    WHEN choice="2" THEN
      CALL job_edit
    ENDWHEN
    WHEN choice="3" THEN
      CALL hours_edit
    ENDWHEN
    WHEN choice="4" THEN
      CALL paypost
    ENDWHEN
    WHEN choice="5" THEN
      CALL check_report
    ENDWHEN
  ENDCASE
ENDLOOP
```

```
PRINT "ENTER YOUR SELECTION :";
choice=GETKEY
IF choice<" " THEN
  PRINT "."
ELSE
  PRINT choice
ENDIF
```

These statements will prompt the user to enter a number corresponding to one of the above PRINTed choices. After a single keystroke is placed in choice, it is echoed to the screen, since GETKEY does not echo its value.

```
IF choice<"0" OR choice>"5" THEN
  REDO
ENDIF
```

If the choice was too low (choice<"0") or the choice was too high (choice>"5") then execution goes back to LOOP.

CASE

CASE marks the start of a series of WHEN ... ENDWHEN statements. Each WHEN has a condition and then some action between the WHEN and the ENDWHEN. What happens is that each condition is tested until a condition is found that is true. When this condition is found the statements between this WHEN ... ENDWHEN are executed, and then execution continues after the matching ENDCASE.

```
WHEN choice="0" THEN
  CLOSE ALL
  QUIT
ENDWHEN
```

This will close all the open files and stop execution.

```
WHEN choice="1" THEN
  CALL employee_edit
ENDWHEN
```

If the choice were 1 then the module to edit the employee_data file, (created in LESSON 4A), is CALLed. CALLing simply means that the module named is executed and when it is finished, execution continues at the statement after the CALL.

```
WHEN choice="2" THEN
  CALL job_edit
ENDWHEN
```

If the choice were 2 then the module to edit the job_data file, (created in LESSON 4B), is CALLed.

```
PAYROLL MAINTENANCE                    January 18, 1986  12:55:39

        0. Quit
        1. Employee Maintainance
        2. Job Maintainance
        3. Time Maintainance
        4. Post Time Reports
        5. Report On Posted Records

ENTER YOUR SELECTION:
```

```
WHEN choice="3" THEN
  CALL hours_edit
ENDWHEN
```
If the choice were 3 then the module to edit the hours_data file, (created in LESSON 4C), is CALLed.

```
WHEN choice="4" THEN
  CALL paypost
ENDWHEN
```
If the choice were 4 then the posting module is called.

```
WHEN choice="5" THEN
  CALL check_report
ENDWHEN
```
If the choice were 5 then the report module, created in LESSON 4D, is called to print out a report.

ENDCASE
This marks the end of the CASE, meaning no more WHEN ... ENDWHEN statements are allowed.

ENDLOOP
This marks the end of the LOOP. Execution will continue at LOOP.

So there is our payroll program. Go to the main menu and press 6 to execute the **payroll** module. If all has gone well, you will get a screen like the one on the opposite page. The first thing to be done is to type in some employee and job data records. Next, type in hours records for those employees on these jobs. Then choose the option to post these reports. Finally, choose the report option to print out the check_data information.

If something does go wrong during execution of one of the modules, so that a module stops with an error message, carefully check the statements you typed in. Most probably you made a typing error in one of the statements, so simply correct it, recompile and try again. These modules have been extensively tested and they do work!

LESSON 5

BACKUPS AND MODIFYING STRUCTURE

Objectives:

- to learn how to backup and change the structure of data files
- to learn the COPY file command

Backups - file structure maintained

COPY (see COPY in reference manual for more details) is used to make backups of existing data files. In the simplest case where data is to go unchanged from one file to its backup file the operation is quite straightforward. You must first open the data file, copy the structure of that data file to an unopened backup file, (if the file exists it will be overwritten), and finally open the backup file and copy all (or some) of the information from the data file to the backup file. For example in IMS interactive mode:

```
OPEN "mail_list"
COPY STRUCTURE OF FILE mail_list TO "mail_listbak"
OPEN "mail_listbak"
COPY FILE mail_list TO FILE mail_listbak
```

This will open the data file **mail_list**, copy its structure to file **mail_listbak**, then open **mail_listbak** and copy the data.

Since at the end of a COPY statement there may be a range specification, the COPY statement could be made into the following:

```
COPY FILE mail_list TO FILE mail_listbak LET
     mail_listbak.address1=
     CAP$(mail_list.address1)
     PRINT name
```

This does the same copy as before except the field **mail_listbak.address1** will be in capital letters and the name field of the record will be printed out as it is copied. Note that the current file in a copy statement is the first file, so the above statement could be written:

```
COPY FILE mail_list TO FILE mail_listback LET
     mail_list.address1 = CAP$(address1)
     PRINT name
```

Modifying Structure

COPY is also used to modify the structure of a file. The procedure is to create a new file with the wanted structure, then use a COPY with LET statements to assign data from the old file to the new fields of the new file.

For example, suppose the mail list file in the previous example didn't have a comment field. It could be added by creating a new file with all the same fields and keys as the old mail list file plus a comment field. Then the following statements would copy the data into the new file:

```
OPEN "mail_list"
OPEN "mail_list2"
COPY FILE mail_list TO FILE mail_list2 LET
     mail_list2.comment ="New Comment"
```

Since **mail_list2** is the new file and it has the same field names as the old **mail_list** file, the contents of these fields are copied as before. However, file **mail_list2** has a new field called **comment**, and this is explictly assigned to in the LET statement.

Adding another key is even simpler. In a COPY statement the keys are assigned automatically; no assignment is needed or allowed. To add or take away keys one would create a file with the same fields and more or fewer keys, open both files and copy the old one to the new one.

```
MODULE error_trap_demo


TEXT a$ OF 25
DATE dt,dt2
INTEGER n
REAL r


SET TRAP TO errortrap
PRINT "Enter a date :";

LABEL date_enter

INPUT a$
dt=DATE(a$)
PRINT "Enter another date :";
INPUT dt2
n=INTEGER(dt-dt2)
PRINT "There are "; ABS(n); " days between the two dates."

PRINT "Enter a number: ";
INPUT r
PRINT "1 divided by ";r;" is ";1/r

OPEN "file1" AS f1
CLOSE f1

END


LABEL errortrap

PRINT "*** ERROR ***"
CASE
  WHEN ERROR=12 THEN
    PRINT "Improper date"
    PRINT "Please re-enter"
    RESUME AT date_enter
  ENDWHEN
  WHEN ERROR=19 THEN
    PRINT "division by zero"
    PRINT "resuming execution"
    RESUME
  ENDWHEN
  WHEN ERROR=42 THEN
    PRINT "Improper file named in OPEN"
    QUIT
  ENDWHEN
  PRINT "Abnormal error"
  HELP ERROR
  RETRY
ENDCASE
```

LESSON 6

Error Trapping

Objectives:

To learn about:

Error trapping statements
- SET TRAP TO
- RESUME
- RESUME AT
- RETRY

other statements
- PRINT
- DATE
- ABS
- QUIT
- END

Error trapping allows the module to handle errors in a straightforward manner. For a full list of errors see the appendix in the reference manual under ERROR NUMBERS.

Basically, the idea of error trapping is that all errors which can occur during execution of a module are assigned a number. These error numbers can be specially tested by an "error trapper", which will then take appropriate action to handle the error.

Enter the text editor with the file name **error.imo**. Type in the statements shown on the opposite page, compile and then execute the program. An explanation follows below:

MODULE error_trap_demo

The name of this module is error_trap_demo.

TEXT a$ of 25

A TEXT variable, called "a$" is declared to have a length of 25 characters.

DATE dt,dt2

Two DATE variables, called "dt" and "dt2", are declared.

INTEGER n

An INTEGER variable, called "n", is declared.

```
MODULE error_trap_demo


TEXT a$ OF 25
DATE dt,dt2
INTEGER n
REAL r


SET TRAP TO errortrap
PRINT "Enter a date :";

LABEL date_enter

INPUT a$
dt=DATE(a$)
PRINT "Enter another date :";
INPUT dt2
n=INTEGER(dt-dt2)
PRINT "There are "; ABS(n); " days between the two dates."

PRINT "Enter a number: ";
INPUT r
PRINT "1 divided by ";r;" is ";1/r

OPEN "file1" AS f1
CLOSE f1

END


LABEL errortrap

PRINT "*** ERROR ***"
CASE
  WHEN ERROR=12 THEN
    PRINT "Improper date"
    PRINT "Please re-enter"
    RESUME AT date_enter
  ENDWHEN
  WHEN ERROR=19 THEN
    PRINT "division by zero"
    PRINT "resuming execution"
    RESUME
  ENDWHEN
  WHEN ERROR=42 THEN
    PRINT "Improper file named in OPEN"
    QUIT
  ENDWHEN
  PRINT "Abnormal error"
  HELP ERROR
  RETRY
ENDCASE
```

REAL r

A REAL variable is declared with the name of "r".

SET TRAP TO errortrap

This is the first of the error trapping statements, and it should come near the start of the module. When this statement is executed it tells IMS that should any kind of error occur, to go to the statement after the label "errortrap". For example, if anywhere in the module a statement divides a number by zero then execution will continue at the statement after the "LABEL errortrap" statement. The "LABEL errortrap" statement must therefore be present in the module.

By the way, if a module has no error trap, (ie. it has no SET TRAP TO statement), and an error occurs then execution stops and IMS prints the error number. Also note that an error trap exists only for that module. This means that a SET TRAP TO statement is in effect only while the module in which it occurs is executing.

PRINT "Enter a date :";

This PRINT statement prompts the operator to enter a date, like "JUNE 23,1985".

LABEL date_enter

This marks the following statement for the error trapper, and its purpose will become obvious later.

INPUT a$

This lets the operator enter a value to respond to the previous prompt message, and the value is stored in a$.

dt=DATE(a$)

Note that the previous statement inputs a value into a variable of type TEXT, not of type DATE. This is only for illustration; the next two lines show an easier way to do this. But when you have a TEXT value (a$), and you want to have it converted to a DATE value (dt), you can use the DATE statement as shown above. Remember that the TEXT value should be in a recognizable date format.

PRINT "Enter another date :";
INPUT dt2

These two statements prompt the operator to enter another date and then it is input. Since the variable in the INPUT statement is of type DATE, then should the user enter a value that is not a DATE, IMS asks the operator to re-enter the value and no error number is generated. An automatic retry is done if the error is in an INPUT statement. The error handler is not entered.

```
MODULE error_trap_demo


TEXT a$ OF 25
DATE dt,dt2
INTEGER n
REAL r


SET TRAP TO errortrap
PRINT "Enter a date :";

LABEL date_enter

INPUT a$
dt=DATE(a$)
PRINT "Enter another date :";
INPUT dt2
n=INTEGER(dt-dt2)
PRINT "There are "; ABS(n); " days between the two dates."

PRINT "Enter a number: ";
INPUT r
PRINT "1 divided by ";r;" is ";1/r

OPEN "file1" AS f1
CLOSE f1

END


LABEL errortrap

PRINT "*** ERROR ***"
CASE
  WHEN ERROR=12 THEN
    PRINT "Improper date"
    PRINT "Please re-enter"
    RESUME AT date_enter
  ENDWHEN
  WHEN ERROR=19 THEN
    PRINT "division by zero"
    PRINT "resuming execution"
    RESUME
  ENDWHEN
  WHEN ERROR=42 THEN
    PRINT "Improper file named in OPEN"
    QUIT
  ENDWHEN
  PRINT "Abnormal error"
  HELP ERROR
  RETRY
ENDCASE
```

n=INTEGER(dt-dt2)

This is an example of how to take the difference between dates, useful in situations like ageing. To do this you subtract the days, use the INTEGER statement to convert the value to an integer number, and store the result in a numeric variable.

PRINT "There are "; ABS(n); " days between the two dates."

This is a PRINT statement that outputs 3 values. "There are " and " days between the two dates." are two values that are output as they look. But a PRINT statement can output any expression, here it outputs ABS(n). This stands for absolute value of the variable n. So, for example, if n were -10 then the PRINT statement will output:

There are 10 days between the two dates.

PRINT "Enter a number: ";
INPUT r

This will prompt the operator to enter a number, then INPUT a number from the operator.

PRINT "1 divided by ";r;" is ";1/r

This is another PRINT statement that outputs more than one value. It outputs "1 divided by " then the value you entered followed by " is " and the value of 1 divided by r (the / symbol stands for divide). For example, if r were equal to 2 then the statement outputs:

1 divided by 2 is 0.5

OPEN "file1" AS f1
CLOSE f1

These two statements open and close a file called file1. Notice that the OPEN statement has an AS clause. The AS clause is optional; if it is present then the file tag becomes the word after the "AS". That is why the next statement is **CLOSE f1** and not **CLOSE file1.**

END

END stops execution of the module. If this module had been called by another module then execution would return back to that module. The purpose of END here is to avoid execution of the following error trapper statements, since they should get executed only if an error happens.

LABEL errortrap

This is the label referred to in the SET TRAP TO statement we saw earlier. This then is the start of the error trapper. Note the way things are arranged in this module. The SET TRAP TO statement follows right after the variable

```
MODULE error_trap_demo


TEXT a$ OF 25
DATE dt,dt2
INTEGER n
REAL r


SET TRAP TO errortrap
PRINT "Enter a date :";

LABEL date_enter

INPUT a$
dt=DATE(a$)
PRINT "Enter another date :";
INPUT dt2
n=INTEGER(dt-dt2)
PRINT "There are "; ABS(n); " days between the two dates."

PRINT "Enter a number: ";
INPUT r
PRINT "1 divided by ";r;" is ";1/r

OPEN "file1" AS f1
CLOSE f1

END


LABEL errortrap

PRINT "*** ERROR ***"
CASE
  WHEN ERROR=12 THEN
    PRINT "Improper date"
    PRINT "Please re-enter"
    RESUME AT date_enter
  ENDWHEN
  WHEN ERROR=19 THEN
    PRINT "division by zero"
    PRINT "resuming execution"
    RESUME
  ENDWHEN
  WHEN ERROR=42 THEN
    PRINT "Improper file named in OPEN"
    QUIT
  ENDWHEN
  PRINT "Abnormal error"
  HELP ERROR
  RETRY
ENDCASE
```

declarations, then comes the body of the module, then an END statement, and then the error trapper.

PRINT "* ERROR ***"**

This statement outputs "*** ERROR ***", a sign that we are in the error trapper and an error really did occur.

CASE

This marks the start of the CASE ... ENDCASE construction. In between the CASE ... ENDCASE statements are several WHEN ... ENDWHEN statements, each tests for a specific error condition and then takes appropriate action.

WHEN ERROR=12 THEN

ERROR holds the number of the error which occurred. If the error number were 12, then the statements between this WHEN and the next ENDWHEN are executed.

```
PRINT "Improper date"
PRINT "Please re-enter"
RESUME AT date_enter
ENDWHEN
```

ERROR number 12 means that a TEXT value being converted to a DATE was not a valid date value. For example, the operator could have typed in "JU 23,1985" in answer to the prompt for a date seen earlier. Since this is not a valid date, "Improper date" and "Please re-enter" are PRINTed. Then the next error trapping word, RESUME AT is executed. "RESUME AT date_enter" means continue execution at the statement after the "LABEL date_enter" statement we saw earlier. This "date_enter" label marks the INPUT statement for the date, so what happens is that if the operator types in a faulty date value, he or she can re-enter it (instead of all execution stopping with an error message). The last statement, ENDWHEN, marks the end of the previous WHEN statement.

```
WHEN ERROR=19 THEN
PRINT "division by zero "
PRINT "resuming execution"
RESUME
ENDWHEN
```

These statements are executed if the error number were 19: the division by zero error. This would happen if the operator entered zero in answer to the prompt to enter a number. "division by zero" and "resuming execution" are PRINTed out. Then the next error trapping statement, RESUME, is executed. This statement resumes execution at the statement right after the error-causing one. So in this case, execution will continue right after the statement in which the error occurred. The last statement, ENDWHEN, marks the end of the previous WHEN statement.

```
MODULE error_trap_demo


TEXT a$ OF 25
DATE dt,dt2
INTEGER n
REAL r


SET TRAP TO errortrap
PRINT "Enter a date :";

LABEL date_enter

INPUT a$
dt=DATE(a$)
PRINT "Enter another date :";
INPUT dt2
n=INTEGER(dt-dt2)
PRINT "There are "; ABS(n); " days between the two dates."

PRINT "Enter a number: ";
INPUT r
PRINT "1 divided by ";r;" is ";1/r

OPEN "file1" AS f1
CLOSE f1

END


LABEL errortrap

PRINT "*** ERROR ***"
CASE
  WHEN ERROR=12 THEN
    PRINT "Improper date"
    PRINT "Please re-enter"
    RESUME AT date_enter
  ENDWHEN
  WHEN ERROR=19 THEN
    PRINT "division by zero"
    PRINT "resuming execution"
    RESUME
  ENDWHEN
  WHEN ERROR=42 THEN
    PRINT "Improper file named in OPEN"
    QUIT
  ENDWHEN
  PRINT "Abnormal error"
  HELP ERROR
  RETRY
ENDCASE
```

```
WHEN ERROR=42 THEN
PRINT "Improper file named in OPEN"
QUIT
ENDWHEN
```

These statements are executed if the error number were 42, which is the "file missing or not a data file" error. This would happen if "file1" in the OPEN statement seen previously were incorrect. In this case "Improper file named in OPEN" would be output and then the QUIT statement is executed. QUIT simply ends all execution. It acts like an END statement except that it will not return to the calling module. The last statement is ENDWHEN, which marks the end of the previous WHEN statement.

```
PRINT "Abnormal error"
HELP ERROR
RETRY
```

These statements are inside the CASE ... ENDCASE statements so they are therefore still part of that structure. They follow all the WHEN ... ENDWHEN statements, so these statements will be executed only if none of the WHEN conditions is true. If the error number is not 12 or 19 or 42 then these statements are executed. The first outputs a message telling the operator that some strange, unaccounted-for error occurred. The HELP statement prints an explanation of the error. The last statement, RETRY, goes back to reexecute the error causing line. This can be a dubious action because it assumes that the error will eventually go away. In some cases, like reading a marginal disk, it might, but in other cases it will never go away.

```
ENDCASE
```

This marks the end of the CASE ... ENDCASE structure started by the previous CASE statement, and it also ends the error trapper.

One final note about the error trap is that if errors occur inside the error trap then the module stops execution and IMS prints out the error number. Also, a SET TRAP OFF command can be used inside the module body to turn off the error trap set previously by a SET TRAP TO statement. Then the error trap will not be executed when an error occurs, but execution will stop and an error number will be printed out.

NOTE: GOTO should not be used to continue execution after an error. Executing a RESUME, RESUME AT or RETRY resets the ERROR number to zero.

LESSON 7

Adding Functions To IMS

Objectives:

To learn about
- adding CALLable functions to your programs
- parameter passing and global fields
- the EXIT command
- multiple modules per file

In this section we will learn about making general-purpose modules or functions that can be used in many applications.

Specifically, in this section we will create a module to calculate the standard deviation of an array of numbers. Remember the formula for standard deviation is the square root of the following:

$$\text{the sum of the values} \quad \frac{(n - \text{average})^2}{\text{number of elements in the array}}$$

where **n** is each of the elements of the array, and **average** is the average value of the elements of the array.

So we can see that we need the ability to take the square root of a value, to take the average of the array, and to square a value. The square root routine is already part of IMS. We will be creating the average and the square or power routines ourselves. They will be called by our variance routine, which is the above formula without the square root. Finally, our standard deviation module will call our variance module and take the square root to give the standard deviation value.

We will do this in a different way from that of previous lessons. There we had one module per file, meaning that the module takes more memory than it needs and is taken out of memory when it has finished execution. This works well enough in our previous lessons, but in this lesson we are making a set of usable statistics functions that would take up too much memory and be too slow if they were each in a separate file. The answer is "multi-modules", which is the IMS term for putting several modules in one file.

```
NOTE a function which calculates the standard deviation of
NOTE   field array NUMBERS with N elements, and returns the
NOTE   answer.

MODULE standard_dev(n)
END SQRT(CALL variance(n))
```

An important thing to remember when using multi-modules is that the file name should be the same as the first module in the file. When IMS searches for a module it will first check to see if the module has already been loaded and if it has not, it will try to find a file with that name in the execution directory. All the modules of a file will be loaded when a multimodule is loaded. If you call a module which has not been previously loaded and has a name different from the name of the file containing it, it will not be found. This problem can be avoided in two ways. Use the SHELL statement to LOAD the file before any of it's modules are used or CALL the first module of the file and don't END that module until the other modules in the file are no longer needed.

In this lesson, the first module will not END until the other modules in the file are no longer needed. Because of this, we will not need to LOAD and UNLINK the modules manually or with the SHELL statement.

The first module is **standard_dev.** Create a file with this name with the text editor and enter the lines of text on the opposite page. An explanation follows:

NOTE a function which calculates the standard deviation of
NOTE field array NUMBERS with N elements, and returns the
NOTE answer.

This is a comment saying what the module is supposed to do and what the parameters mean. Notice that a field array is being used. A file with this array must be previously OPENed.

MODULE standard_dev(n)

The module's name is **standard_dev,** with parameter **n.** We have seen parameters before, for example ABS(n) has the parameter **n.** Parameters can not be arrays, although they can be array elements. The type of parameter **n** is the same as in the calling module.

END SQRT(CALL variance(n))

The standard deviation module is written in one line - END returns the square root (SQRT) of the value returned by the **variance** module.

```
NOTE a function which calculates the variance of field array
NOTE   NUMBERS, with N elements, and returns the answer

MODULE variance(n)


INTEGER count
REAL mean,diff,sum,square


mean=CALL average(n)
sum=0
count=1
WHILE count<=n DO
  diff=NUMBERS(count)-mean
  CALL power(diff,2,square)
  sum=sum+square
  count=count+1
ENDWHILE
END sum/(n-1)
```

To the text already in the editor add the statements on the opposite page. This module will calculate the variance.

NOTE a function which calculates the variance of field array
NOTE NUMBERS, with N elements, and returns the answer
This is a comment telling what the purpose of the module is, and what the parameters are.

MODULE variance (n)
This names the module as **variance**, with parameter **n**.

INTEGER count
REAL mean,diff,sum,square
The **count** variable keeps track of the number of iterations through the loop. **mean** is used to store the value returned by module **average**, **diff** stores the value of the difference between each array element and the average, **square** stores the value returned from the module **power**, and **sum** holds the running total of the numerator in the aforementioned standard deviation formula.

mean=CALL average(n)
The **average** module which we have not yet created is called in this manner. The average value will be assigned to variable **mean**.

sum=0
count=1
The **sum** and **count** variables are initialized.

WHILE count<=n DO
This starts the loop, which will execute **n** times.

diff=NUMBERS(count)-mean
CALL POWER(diff,2,square)
sum=sum+square
First the difference between the array element and the mean is stored in **diff**. Then **diff** is squared and stored in **square**, which is then added to the running total **sum**. The lines **CALL POWER(diff,2,square)** and **sum=sum+square** could be written as the single line **sum=sum+CALL POWER(diff,2)**, but this would require a different version of the **POWER** module. Try implimenting this suggestion.

count=count+1
count keeps the number of times through the loop.

ENDWHILE
This marks the end of the WHILE loop.

END sum/(n-1)
The module returns the value of **sum** divided by **n-1**.

```
NOTE a function which calculates the average of field array
NOTE   NUMBERS, with N elements, and returns the answer

MODULE average(n)


INTEGER count
REAL sum


count=1
sum=0
WHILE count<=n DO
  sum=sum+NUMBERS(count)
  count=count+1
ENDWHILE
END sum/n
```

Add to the text in the text editor the statements on the opposite page. This is the module that calculates the average.

NOTE a function that calculates the average of field array
NOTE NUMBERS, with N elements, and returns the answer
This is a note telling what the module is supposed to do, and what the parameters mean.

MODULE average(n)
The module is named **average** and has parameter **n.**

INTEGER count
REAL sum
The **count** variable keeps track of how many times to go through the WHILE ... ENDWHILE loop. **sum** is a variable that keeps a running total of the numbers.

count=1
sum=0
Initialize the variables to their starting values.

```
WHILE count<=n DO
  sum=sum+ NUMBERS(count)
  count=count+1
ENDWHILE
```
This is the complete WHILE ... ENDWHILE loop. It will be executed **n** times (the number of elements in the array), and adds up the total of the elements in field array **NUMBERS.** Notice that **NUMBERS** is not declared in this module. It is a field array and field arrays are "global" to all modules after the file containing the field is OPENed. This module must be CALLed by a module that OPENs a file containing a NUMBERS field array, otherwise an error will occur. Remember that arrays cannot be passed as parameters, so we are using a field array in this manner.

END sum/n
This is the standard END statement with an expression following it (the "/" symbol means divide). Any END can be followed by an expression; this means that the module "returns" the value of the expression. END acts the same way as it did before; the module will stop execution at this point and return to the module which called it. The expression means that this module becomes a value. We can go **PRINT CALL average(n), B= CALL average(n)+1. CALL average** will always return the value of **sum/n.** In this way we can avoid using an extra parameter to return the result.

```
NOTE calculates the value of BASE to the integer power of
NOTE   EXPONENT and returns the answer in RESULT

MODULE power(base,exponent,result)


INTEGER count


count=1
result=1
WHILE count<=exponent DO
  result=result*base
  count=count+1
ENDWHILE
END
```

Next is the square or power module. Enter the text on the opposite page. An explanation of each statement follows below:

NOTE calculates the value of BASE to the integer power of
NOTE EXPONENT and returns the answer in RESULT

This is a comment defining the action of the module and what the parameters are to do.

MODULE power(base,exponent,result)

This module has the name **power**, and three parameters - **base**, **exponent**, and **result**. Note that they are surrounded by open and close brackets and they are separated by commas. The above three parameters mean that the module expects three values to come into the module when it is called. These values will be used by the module. It is important to remember that these values can be changed in the module and they return as changed values to the calling module. Module **power** will calculate the value of **base** to the power of **exponent**, and store that value in **result**. The calling module calls POWER with **base** and **exponent** values and uses **result** as the answer. Non parameter variables used in the module are local to that module; they have no effect on variables in any other module. This all means that a module, like **power**, can be written so that another module can CALL it and not worry about it's variables being inadvertently changed.

INTEGER count

This is a variable to help count the number of times through the loop.

count=1
result=1

This initializes the variables to the starting values.

WHILE count<=exponent DO

This starts the loop. It will execute **exponent** times.

result=result*base

We are calculating the value of **base** times itself **exponent** times and saving the value in **result**.

count=count+1

count is the number of times through the loop, and is updated here.

ENDWHILE

This marks the end of the WHILE loop.

END

The end of the module.

```
NOTE this file contains a list of numbers for testing the
NOTE   standard deviation module

FILE numbers_data
FIELD REAL numbers(50)
```

Save the text file, and from the main menu choose option 5 (compile). Answer the prompt with **standard_dev.**

It is now time to test our standard deviation functions. Type in the file descriptor on the opposite page. This file descriptor will create a file to store the numbers for our standard deviation module to process.

An explanation of the lines follows:

NOTE this file contains a list of numbers for testing the
NOTE standard deviation module
This is a comment explaining the importance of the file.

FILE numbers_data
The file will be called "numbers_data".

FIELD REAL numbers(50)
This is a field array of REALs, called NUMBERS. The parentheses after NUMBERS signify that it is an array, and the 50 means that there are 50 elements in the array. In other words, our standard deviation module will be calculating with up to 50 values.

Go ahead and save the file, then use option 2 to generate the file.

```
MODULE standard_dev_test


INTEGER cn


OPEN "numbers_data"
cn=1
PRINT "Enter the numbers, one per line: "
WHILE cn<=50 DO
  INPUT NUMBERS(cn)
  IF NUMBERS(cn)=-9999 THEN
    EXIT
  ENDIF
  cn=cn+1
ENDWHILE

PRINT CALL standard_dev(cn-1)
END
```

Create a text file called **standard_dev_test** and type in the statements on the opposite page. This module will input a range of numbers from the keyboard and when the ending number -9999 is entered, it will call our standard deviation module and print the result. An explanation of the statements follows:

MODULE standard_dev_test
This module is called **standard_dev_test.**

INTEGER cn
This is an INTEGER variable, called **cn.** Its job is to count how many times we have been through the WHILE loop.

OPEN "numbers_data"
This opens the file with the number array in it. You see there is no AS part in this OPEN statement; so **numbers_data** becomes the file tag.

cn=1
This initializes the variable to 1.

PRINT "Enter the numbers, one per line: "
This is a prompt to tell the user to start typing in the numbers.

WHILE cn<= 50 DO
This is a WHILE loop, that will be executed once for each of the 50 elements in the NUMBERS field array (or when the EXIT statement is executed below).

INPUT NUMBERS(cn)
This inputs a value from the keyboard into the field array NUMBERS, in the file **numbers_data.** NUMBERS stores the data for our standard deviation module.

```
IF NUMBERS(cn)= -9999 THEN
  EXIT
ENDIF
```
These statements say that if the value just entered was equal to -9999 then EXIT the loop. EXIT will cause a jump out of the WHILE loop to the statement after the ENDWHILE. Typing -9999 as your final number signifies that data entry is finished. This -9999 value is called a sentinel.

cn=cn+1
The **cn** variable is incremented to keep track of the number of times we have gone through the WHILE loop.

ENDWHILE
This marks the end of the WHILE loop.

```
MODULE standard_dev_test


INTEGER cn


OPEN "numbers_data"
cn=1
PRINT "Enter the numbers, one per line: "
WHILE cn<=50 DO
  INPUT NUMBERS(cn)
  IF NUMBERS(cn)=-9999 THEN
    EXIT
  ENDIF
  cn=cn+1
ENDWHILE

PRINT CALL standard_dev(cn-1)
END
```

PRINT CALL standard_dev(cn-1)

This calls our **standard_dev** module with the value of **cn-1**. If **standard_dev** is not in memory, our library file of standard deviation functions will be loaded. We are passing the value of **cn-1** as a parameter rather than **cn** because the WHILE LOOP actually finishes with **cn** at one more than we want. The value returned by **standard_dev** is printed to show us the standard deviation of our set of numbers.

END

The end of the module.

Save, compile and execute this module. When the prompt appears:

```
Enter the numbers, one per line:
```

the cursor will be waiting for a list of numbers from you. Type in any list of numbers, for example:

```
10.23
14.1
13.34
4.78
6.8332
3.6590
-9999
```

which will output the number:

```
4.0249578279365
```

A side note here is that CALLing modules is always slower than having subroutines, using GOSUB and RETURN, inside the module. For small jobs GOSUB and RETURN, (see the reference manual), are a better choice.

INFORMATION MANAGEMENT SYSTEM

REFERENCE

Clearbrook Software Group

CLEARBROOK SOFTWARE GROUP
INFORMATION MANAGEMENT SYSTEM

REFERENCE MANUAL

Release B
January 1, 1986

TABLE OF CONTENTS

STATEMENTS, FUNCTIONS and KEYWORDS

Input related

Miscellaneous

Numeric functions

Output related

Program control

TEXT functions

Blank

ABS

USAGE:

```
ABS(n)
```

where **n** is any numeric expression.

ABS is a function that returns a number and may only be used in a numeric expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return the absolute value of a number or a numeric expression. If **n** is negative then ABS(**n**) returns the positive value of **n**, if **n** is positive or zero then ABS(**n**) returns **n**. ABS is useful for taking the difference of two numbers. A subtraction may return a negative value, but ABS will always return a positive value -- the "distance" between the two numbers.

EXAMPLE:

```
a=-6
PRINT ABS(a)
PRINT ABS(4)
PRINT ABS(a-3)
```

will output:

```
6
4
9
```

AND

USAGE:

condition1 AND **condition2**

where **condition1** and **condition2** are both any expressions that evaluate to true or false. Examples of conditional expressions are **amount<total** and **balance>deductions**, which all have relationals like <, >, =, <=, >=, **CT**, etc. in them (see RELATIONALS).

AND may only be used between two conditions to make up one larger condition, and therefore can only be used in places where a condition is allowed, -- in range specifications as well as in IF, WHILE, and UNTIL statements.

PURPOSE AND OPERATION:

To test for cases when both the first condition and the second condition are true. AND returns true if both conditions are true; if neither is true or only one is true then AND returns false. ANDs can be used with ORs, NOTs, & XORs to build large conditions made up of several smaller conditions.

EXAMPLE:

```
IF amount>net AND amount<gross THEN
  PRINT "amount is in range"
ELSE
  PRINT "amount is out of range"
ENDIF
```

If **amount** is greater than **net** and **amount** is less than **gross** then it will output:

```
amount is in range
```

and if either condition is false then

```
amount is out of range
```

is PRINTed out.

arrays

USAGE:

```
TEXT identifier (dim list) [OF [LENGTH] number]
INTEGER identifier (dim list)
LONG INTEGER identifier (dim list)
DATE identifier (dim list)
REAL identifier (dim list)
```

where **dim list** is a list of numbers, each one separated by a comma.

Also note that each of these declarations can be preceded by FIELD in a file descriptor, thus becoming field arrays.

PURPOSE AND OPERATION:

To allow for easier management of related items. An array is simply a collection of similar data items; the entire collection is called the **array**, and an individual item is called an **element**. Each element is referenced by the **index**, which is a number in the range of the array. An array may have more than one **dimension**, a one-dimensional array would be a list, a two-dimensional array would be a table, a three-dimensional array would be a table with another parameter such as time, etc. For example:

```
TEXT$ series_no(12) of 20
```

would be an array or list of 12 series_no elements, series_no(1), series_no(2), series_no(3), ... series_no(12); each element is a TEXT value of 20 characters. Another example:

```
INTEGER stats(5,20)
```

would be an array or table of 5 rows with 20 columns; a total of 5x20 = 100 elements altogether. The elements are stats(1,1), stats(1,2), stats(1,3), ..., stats(1,20), stats(2,1), stats(2,2), stats (2,3), ..., stats(2,20), stats(3,1), stats(3,2), and so on up to stats(5,20).

EXAMPLE:

```
INTEGER house_numbers(200)
TEXT response$(23) of 30
REAL pay_rates(10)
PRINT house_numbers(2)
PRINT pay_rates(5)
```

declares 3 arrays: an INTEGER array of 200 house numbers, a TEXT array of 23 responses, each of 30 characters, and a REAL array of 10 pay rates. Then it will output the second house number and the fifth pay rate.

ASCII

USAGE:

```
ASCII($)
```

where $ is any TEXT expression.

ASCII is a function that returns a number and may only be used in a numeric expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return the ASCII value of the first character in $ (see the ASCII table in the appendix). ASCII has a variety of miscellaneous uses; it is useful when text values are uncertain and may contain control codes, strange characters, etc.

EXAMPLE:

```
a$="An apple for your thoughts."
print ASCII(a$)
```

will output:

```
65
```

which is the ASCII value of "A".

CALL

USAGE:

CALL **modulename (list of parameters)**
or
CALL **modulename**

where **list of parameters** is any number of expressions consisting of variables, TEXT values, and numbers - each separated by a comma.

CALL may also be used in an expression in which case it has the value of the END expression in the called module.

```
LET a = CALL ...
```

PURPOSE AND OPERATION:

To call another module. CALL is followed by a module's name and then an optional list of parameters. If a calling parameter is a variable or field name, any change to that parameter in the called program will affect the variable or field in the calling program.

For example, module1 could have the following statements:

```
a=1
b=a+56
c=3
CALL module2(a,b,c+0)
```

There are three parameters in this CALL statement, **a** with a value of 1, **b** with a value of 57, and **c+0** with a value of 3. Now, module2 would also have to be declared with 3 parameters, for example:

```
MODULE module2(first,second,third)
```

because it is being CALLed with three parameters. Note that parameter **first** gets the value of **a**, **second** gets the value of **b**, and **third** gets the value of **c+0**. The order of the parameters in the CALL and MODULE declarations is very important. If the following statements appeared in module2:

```
first=first+1
second=second-1
third=0
END
```

a would have a value of 1 before the CALL statement, and a value of 2 after. **b** would have a value of 57 before and 56 after. **c** will remain the same because it was passed as part of an expression.

CALL is very useful when you have a module that does one particular job which can be used by other modules. This

makes for more maintainable and easier to understand programs. It is important to remember that the called module can only change those variables in the calling module which are used in the parameter list. Finally, IMS fully supports recursion, ie., a module can call itself.

EXAMPLE:

1.
```
MODULE caller
CALL submod
PRINT "in caller"
CALL submod
END

MODULE submod
PRINT "in submod"
END
```

submod will output:
```
in submod
in caller
in submod
```

2.
```
MODULE caller
INTEGER a
TEXT a$,b$,c$ of 255

a=5
a$="hello there"
PRINT "in caller"
CALL submod(a,a$,30+3)
PRINT "back in caller"
PRINT a
END


MODULE submod(first,second$,third)
INTEGER a
PRINT "in submod"
PRINT first,second$,third
first=first+1
a=10
END
```

will output:
```
in caller
in submod
5          hello there    33
back in caller
6
```

Module CALLER calls module SUBMOD with parameters **a**, **a$**, and **30+3**. These three values come into SUBMOD and are assigned to **first**, **second$**, and **third**. Note that these three are not declared in module SUBMOD. Also note that variable **first** changes value in SUBMOD. Then when SUBMOD is finished it returns the values of **first**, **second$**, and **third** back to module CALLER's parameter list - **a**, **a$**, and **30+3**. That is why the value of **a** has changed from 5 to 6. Finally, note that SUBMOD has a variable called **a** but the use of **a** has no effect on the variable **a** in module CALLER. Only the parameters can return with changed values.

3.
```
MODULE current_date
END "The current date is: "+today

MODULE caller
PRINT CALL current_date
```

Here module CURRENT_DATE has an END statement followed by an expression. This means that CURRENT_DATE becomes a function which can be used in any expression. Module CALLER PRINTs out the value that CURRENT_DATE returns, which would be something similar to:

```
The current date is: June 23,1985
```

CAP$

USAGE:

```
CAP$($)
```

where $ is any TEXT expression.

CAP$ is a function that returns a TEXT value and may only be used in a TEXT expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return a TEXT value with all lower case letters converted to upper case. CAP$ is useful for handling text which may be a mixture of lower and upper case, but which should be seen as equivalent.

EXAMPLE:

```
name1$="John Doe"
name2$="JOHN DOE"
IF CAP$(name1$)=CAP$(name2$) THEN
  PRINT "SAME NAME"
ENDIF
```

will output:

```
SAME NAME
```

CASE ... WHEN ... ENDWHEN ... ENDCASE

USAGE:

```
CASE
 ...
  WHEN condition THEN
     ...
  ENDWHEN
 ...
ENDCASE
```

There may be as many WHEN ... ENDWHEN statements within the CASE ... ENDCASE as necessary.

PURPOSE AND OPERATION:

To make multi-way branches simpler. CASE ... ENDCASE marks the start and end of the branches, and multiple WHEN ... ENDWHENs define each individual branch. WHEN is followed by **condition** and THEN. **Condition** is any expression that evaluates to true or false. Examples are **amount<total** and **balance>deductions** which all have relationals like <, >, =, <=, >=, **CT**, etc. in them (see RELATIONALS). Statements between the CASE and the first WHEN are always executed, only one WHEN ... ENDWHEN is executed (the first true condition), and if no condition is true then the statements between the last ENDWHEN up to ENDCASE are executed. CASE ... ENDCASE are therefore handy for testing ranges; situations where low, mid, and high ranges are different and must be handled separately.

EXAMPLE:

```
a=45
WHILE a<=105 DO
  CASE
    PRINT a; " ";
    WHEN a<50 THEN
      PRINT "fail"
    ENDWHEN
    PRINT "You passed!! ";
    WHEN a<65 THEN
      PRINT "pass"
    ENDWHEN
    WHEN a<80 THEN
      PRINT "second class"
    ENDWHEN
    WHEN a<=100 THEN
```

```
            PRINT "first class"
          ENDWHEN
          PRINT "someone is cheating!"
        ENDCASE
        a=a+5
      ENDWHILE
```

will output:

```
45 fail
50 You passed!! pass
55 You passed!! pass
60 You passed!! pass
65 You passed!! second class
70 You passed!! second class
75 You passed!! second class
80 You passed!! first class
85 You passed!! first class
90 You passed!! first class
95 You passed!! first class
100 You passed!! first class
105 You passed!! someone is cheating!
```

The WHILE loop increments the integer **a** by 5 from 45 to 105. In each iteration of the loop it executes the CASE ... ENDCASE statement, prints the value of **a**, and then prints a message based on the value of **a**. If the value is 50 or greater then the following words are PRINTed on the screen:

```
You passed!!
```

Note that when **a** is 105, none of the WHEN conditions is true, so

```
someone is cheating!
```

is PRINTed out.

CHAIN

USAGE:

CHAIN **modulename (list of parameters)**
or
CHAIN **modulename**

where **modulename** is the name of the program module and **list of parameters** is any number of expressions, consisting of variables, TEXT values, and numbers, -- the individual parameters separated by commas.

PURPOSE AND OPERATION:

To GOTO another module, taking the current module out of memory and starting execution at the beginning of the called module. Unlike CALL, CHAIN does not return to the calling module. CHAIN is useful in cases where the memory used by all the modules exceeds the memory available. It allows you to separate the problem into different modules, and then have a minimum number of modules in memory at any one time.

EXAMPLE:

```
MODULE menu
INTEGER choice
  ...
PRINT "1. Enter transactions"
PRINT "2. Print transactions"
PRINT "3. Post transactions"
  ...
PRINT "Enter number of your choice: ";
INPUT choice
CALL prog(choice)

[another file]

MODULE prog (selection)
  ...
CASE
  WHEN selection =1 THEN
    CHAIN enter_transactions
  ENDWHEN
  WHEN selection =2 THEN
    CHAIN print_transactions
  ENDWHEN
  WHEN selection =3 THEN
    CHAIN post_transactions
```

```
        ENDWHEN
          ...
      ENDCASE
```

This series of modules forms a skeleton for a data processing job. Module MENU prints out a list of choices for the user to select, the user selects one and module PROG is CALLed. PROG does some initialization work, and then CHAINs to the various modules that do the individual tasks. PROG will then disappear from memory, and the CHAINed module will come into memory. When that module is finished, execution will go back to MENU, just after the place PROG was called. Note that MENU could call the individual tasks directly, but this way the initialization work of PROG is not in memory when it is not needed.

CHD

USAGE:

CHD **pathlist**

where **pathlist** is a TEXT expression.

PURPOSE AND OPERATION:

To change the working directory. The working directory is where new files are stored and files are expected to be. This action can also be done from the main menu.

EXAMPLE:

```
CHD "/d0/accounts"
OPEN FILE "Smith"
  ...
CLOSE Smith
CHD "/d0/data"
OPEN FILE "ytd_totals"
  ...
CLOSE ytd_totals
```

This will open and close the file called **Smith** in the **/d0/accounts** directory, then it will open and close the file called **ytd_totals** in the **/d0/data** directory.

CHECK

USAGE:

CHECK **file tag key clause**

where **file tag** and **key clause** are optional and refer to an existing key in an already OPENed file (see KEY CLAUSE).

PURPOSE AND OPERATION:

To check the integrity of a file. CHECK will output a report stating whether the file is GOOD or BAD (needing a REINDEX). CHECK is useful when the file is suspect; perhaps the modules working with it are reporting incorrect data.

EXAMPLE:

```
NOTE "mail_list" has fields NAME and ADDRESS
NOTE it's key is called NAME
OPEN "mail_list" as MAIL
CHECK
```

would output something like:

```
Checking deleted record chain of MAIL-GOOD
Index NAME-GOOD
```

CHR$

USAGE:

```
CHR$(n)
```

where n is any numeric expression that evaluates to a value of 0 to 255.

CHR$ is a function that returns a TEXT value and may only be used in a TEXT expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return the character that the ASCII value of **n** represents (see ASCII table in appendix). CHR$ is useful in miscellaneous situations including PRINTing control codes to screens and printers.

EXAMPLE:

```
a=72
PRINT CHR$(a)
```

will output:

```
H
```

which is the ASCII character value for 72.

CLEAR

USAGE:

CLEAR **file tag** **key clause**

where **file tag** and **key clause** are optional and refer to an existing key in an already OPENed file.

PURPOSE AND OPERATION:

To clear a file and its indexes. CLEAR forgets all data in the stated file, (or current file if none is stated). The file structure remains intact.

EXAMPLE:

```
OPEN "temp_data" AS TEMP
CLEAR FILE TEMP
```

CLEAR FORM

USAGE:

CLEAR FORM

PURPOSE AND OPERATION:

To clear the fields present on the form, both on the screen and in the current record. CLEAR FORM is useful in editing a file using a screen form, and can only be used when a screen form is in effect (see the SET FORM TO statement).

EXAMPLE:

```
SET FORM TO "maillist"
ENTER name
ENTER address
 ...
PRINT "Next record ? (Y/N): ";
INPUT response$
IF response$="Y" THEN
  CLEAR FORM
ENDIF
```

CLEAR LINE

USAGE:

```
CLEAR LINE
```

PURPOSE AND OPERATION:

To clear all the text on the current line of the screen. CLEAR LINE is useful when printing out a single-line error message, prompt, etc., and the line should be later cleared from the screen.

EXAMPLE:

```
PRINT "ERROR: INVALID DATA (press any key)";
A$=GETKEY
CLEAR LINE
 ...
```

This will output to the screen:

```
ERROR: INVALID DATA   (press any key)
```

The module will wait until a key is pressed before continuing execution. Then it will blank out the error message line from the screen.

CLEAR SCREEN

USAGE:

```
CLEAR SCREEN
```

PURPOSE AND OPERATION:

To clear the screen. CLEAR SCREEN comes in handy in displaying menus, reports, etc.

EXAMPLE:

```
CLEAR SCREEN
PRINT
PRINT "MENU FOR DATA PROCESSING "
 ...
```

This will clear the screen before the menu is PRINTed on screen.

CLOSE

USAGE:

```
CLOSE file tag
CLOSE
CLOSE ALL
```

where **file tag** is the tag of a previously OPENed file.

PURPOSE AND OPERATION:

To close a file. CLOSE indicates that you no longer need information from the file. All files which are OPENed must be later CLOSEd to ensure that data is written to the file (normally when a program quits, files are closed automatically). If **file tag** is not specified then the current file is closed. If ALL is specified then all OPENed files are closed.

EXAMPLE:

```
OPEN "vendor_list" as VENDORS
CLOSE FILE VENDORS
```

will OPEN and CLOSE file **vendor_list.**

constant

USAGE:

```
TEXT constant     "t"
    or            't'
INTEGER constant  d
REAL constant     d.d
    or            d.
    or            d.dEe
DATE constant     "date"
```

where **t** is any text, **d** is a series of digits and **e** is an integer number between -64 and 63.

PURPOSE AND OPERATION:

To provide constants in a module. Constants are simply values that are written out explicitly. TEXT constants are enclosed by double or single quotes, INTEGER constants are numbers with no decimal point or exponent and REAL constants are numbers with a decimal point or exponent. Therefore, **7. / 4.** are two REALs that return the REAL amount 1.75, while **7 / 4** are two INTEGERs that return the INTEGER amount 1 (See OPERATORS for a discussion on REAL versus INTEGER division).

EXAMPLE:

```
PRINT "this 'is' text"
PRINT 'and "more" text'
PRINT 7/4
PRINT 7./4.
```

will output:

```
this 'is' text
and "more" text
1
1.75
```

USAGE:

COPY **file tag1 key clause** TO **file tag2 range**
or
COPY STRUCTURE OF **file tag1 key clause** TO **filename**

where **file tag1, key clause** and **range** are optional and refer to an existing key in an already opened file and **range** refers to the range specification (see RANGE and KEY CLAUSE). **file tag2** is the tag of an open file and **filename** is a text expression naming the file to which the structure should be copied.

PURPOSE AND OPERATION:

To copy a range of records from one file to another. COPY will go through the specified file by the specified key, (or through the current file and key if none is specified), and look for each record that matches the **range** specification. When such a record is found, the fields with the same name in both of the files are copied to the "TO file". If **range** is not specified then all records are copied. No checks are made to see if the records already exist in the "TO file"; in that case multiple copies of the record will appear. Assignments are allowed to fields in the **range** specification; they will take effect on fields in the "TO file".

COPY STRUCTURE is used to create a new file with the same structure. If the file exists, COPY STRUCTURE will first delete all data in the file, then copy the structure as before.

EXAMPLE:

```
NOTE "mail_list2" has fields NAME, ADDRESS1,
NOTE  ADDRESS2, PHONE, ZIP, and CURRDATE
NOTE  with NAME as a key

OPEN "mail_list2" AS MAILOLD
COPY  STRUCTURE  OF  FILE  MAILOLD  TO
     "MAIL_LIST2BACKUP"
OPEN "mail_list2backup" as MAILNEW
COPY FILE MAILOLD KEY NAME TO FILE MAILNEW ALL FOR
     ADDRESS2 <> "Des Moines, Iowa" PRINT NAME LET
     CURRDATE=TODAY
CLOSE FILE MAILNEW
CLOSE FILE MAILOLD
```

will copy all records from file **mail_list2** to **mail_list2backup**, (except for those with address "Des Moines, Iowa"), giving a list of the names copied, and changing the CURRDATE field to today's date in **mail_list2backup.**

data types

USAGE:

TEXT **identifier** [OF [LENGTH] **number**]
INTEGER **identifier**
LONG [INTEGER] **identifier**
REAL **identifier**
DATE **identifier**

PURPOSE AND OPERATION:

To express what kind of value the **identifier** variable is. The choices are:

NONNUMERIC TYPES

TEXT maximum declarable length = 32767

This is a sequence of characters. The characters can be digits, letters, control codes, punctuation, etc., and they have a maximum length of 32767 characters. TEXT values are not numbers; they cannot be added, subtracted, or tested as numbers. See VALUE for the function to convert a TEXT value into a number. TEXT is useful for storing names, addresses, alphabetic series codes, etc. The OF **number** part states the maximum number of characters that the TEXT variable can hold. The default maximum length is 40 characters.

DATE AD January 1,1 to (about) AD 5000

This is a date that cannot start earlier than January 1,1. DATE is useful for all situations where time is important: on invoices, aging periods, payroll periods, etc. DATE can be formatted in many different ways, see MASK.

NOTE: Because of changes to the calendar over the years, very old dates are not accurate (current rules are used).

NUMERIC TYPES

INTEGER -32768 to 32767

This is a number in the range -32768 to 32767. No decimal places are allowed. INTEGER is useful for storing the number of something - employees, aging days, items in stock, etc. The value has a maximum of 32767; going higher causes an error. INTEGER has the advantage over LONG INTEGER of needing only half the memory or disk space.

LONG INTEGER or **LONG** -2147483648 to 2147483647

This is a whole number in the range -2147483468 to 2147483647. LONG INTEGER is useful for situations where the maximum range on an INTEGER would be too low, Examples are: number of widgets produced last year, number of records in the file, number of users in the area, etc.

REAL -9.9999999999999E+63 to 9.9999999999999E+63
(the exponent has a range of -64 to +63)

This is a number that allows decimal places. Dollars and cents, fractions, percentages, etc., are all examples of numbers which require the REAL data type. REAL also allows an exponent, to express the number in "scientific notation". For example:

12.34E+10

is short for 12.34 x 10^{10} (ten raised to the power 10), which is equal to 123400000000. REALs have up to 14 decimal places; the 14th digit is rounded up from the 15th.

EXAMPLE:

```
TEXT a$ OF 80
TEXT RESPONSE$ OF LENGTH 80
LONG number_of_records
INTEGER number_of_employees
REAL balance,cost
DATE dt
```

DATE

USAGE:

DATE(**expression**)

DATE is a function that returns a date and may only be used in a DATE type expression (see EXPRESSION).

PURPOSE AND OPERATION:

To convert an expression into a date value.

EXAMPLE:

```
date dt
integer n
SET DATE TO "Y/n/d"
NOTE  no DATE function needed here
dt="June 23,1985"

PRINT dt
PRINT dt+7
PRINT dt-7
n=DATE("Sept 1,1985")-dt
PRINT n
```

will output:

```
1985/6/23
1985/6/30
1985/6/16
70
```

DELETE

USAGE:

DELETE **file tag key clause range**

where **file tag**, **key clause** and **range** are all optional and refer to an existing key in an already OPENed file and **range** is a range specification (see RANGE and KEY CLAUSE).

PURPOSE AND OPERATION:

To delete a range of records. DELETE will go through the specified file by the specified key (or through the current file and key if none is specified), deleting each record that matches the **range** specifications. If **range** is not specified then only the current record is DELETEd.

EXAMPLE:

```
NOTE file "salesmen_list" has field NAME, SALES
OPEN "salesmen_list" AS SMAN
PRINT "Deleting useless salesmen"
DELETE ALL FOR SALES <100.00 PRINT NAME
```

will output:

```
Deleting useless salesmen
```

and then go through the entire salesmen_list file, deleting all records with SALES less than 100.00 and printing out the names.

DISPLAY

USAGE:

```
DISPLAY field name
  or
DISPLAY field name MASK $
```

where **field name** is the name of a field and **$** is a TEXT value giving a MASK specification (see MASK).

PURPOSE AND OPERATION:

To show the contents of a field on the form. The SET FORM TO commands must have been previously used to specify which form to use. The DISPLAY **field name** command will show the contents of the named field in the current record. If no MASK value is specified then the mask used is the one specified in the file creation process for that field.

EXAMPLE:

```
OPEN "mail_list"
SET FORM TO "mail_listform"
DISPLAY NAME
DISPLAY ADDRESS1
DISPLAY ADDRESS2
DISPLAY CODE MASK "L#L-#L#"
```

will open the mail list file, show the form on screen, and display the values of the NAME, ADDRESS1, ADDRESS2 fields on the form using the masks given in the file descriptor to create the file. The CODE field will be displayed with the MASK specified.

DUPLICATE

USAGE:

DUPLICATE(**expression**)

PURPOSE AND OPERATION

To return TRUE if **expression** is found in the current key of the current file. DUPLICATE is a function that returns a number and may only be used in a numeric or conditional expression (see EXPRESSION).

EXAMPLE:

```
OPEN "maillist"
USE Key name
PRINT "NAME:";
INPUT name
IF DUPLICATE(name) THEN
   PRINT "That name already exists."
ELSE PRINT "ADDRESS:";
     INPUT address
     PRINT "CITY:";
     INPUT city
     PRINT "COUNTRY:";
     INPUT country
INSERT
ENDIF
```

EJECT PAGE

USAGE:

```
EJECT PAGE
```

PURPOSE AND OPERATION:

To start a new page on the printer. Specifically, EJECT PAGE will send a form feed character to the alternative print path, which is the path named in the SET PRINT TO statement. This will cause the printer or alternative print path file to move the paper to the start of a new page.

If a footer trap has been SET, the footer will be printed at the bottom of the page.

EXAMPLE:

```
SET PRINT TO "/P"
EJECT PAGE
PRINT "Here is the report"
 ...
```

This will set output to go to the **/P** device, (the printer), cause a new page to be started on the printer, and type

```
Here is the report
```

on the printer.

END

USAGE:

```
END
  or
END expression
```

PURPOSE AND OPERATION:

To end the execution of the current module. END will stop execution of the current module, and return to the calling module. If no module called the current one, you return to the process which invoked IMS. This makes END different from QUIT, because QUIT never returns to the calling module.

expression can be any kind of expression, and specifies that the module return a value to the calling module. In effect, the called module becomes a function returning a value that can be PRINTed out, used in a range specification, assigned to a variable, etc. If no value follows END, 0 is defaulted.

EXAMPLE:

1.
```
INPUT response$
IF response$= "Y" THEN
  CALL report
ELSE
  END
ENDIF
```

which will input **response$** and call module REPORT if **response$** is **Y**, otherwise it will terminate execution of the module.

2.
```
MODULE full_date
END today+"  "+time

MODULE calling
PRINT CALL full_date
```

will output:

```
June 23, 1985  10:23:41
```

ENTER

USAGE:

```
ENTER field name
  or
ENTER field name MASK $
```

where $ is a text expression overriding the default mask for the given field name.

PURPOSE AND OPERATION:

To enter a value into a field of a form. The file containing the field must be OPEN and the SET FORM TO command used to specify which form is to be used. The MASK part is optional; if it is present then the field is entered using the specified mask. If it is not present then the mask specified in the file descriptor is used.

EXAMPLE:

```
OPEN "mail_list"
SET FORM TO "mail_listform"
ENTER NAME
ENTER ADDRESS1
ENTER ADDRESS2
ENTER CODE MASK "L#L-#L#"
```

will open the mail list file, display its form on the screen, and enter values into the NAME, ADDRESS1, and ADDRESS2 fields using the masks specified in the file descriptor. Then it will enter a value into the CODE field using the specified mask.

EOF

USAGE:

EOF (**file tag**)

EOF is a function that returns a number and may only be used in a numeric or conditional expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return a value indicating if the stated file is past the last record of the file. EOF stands for End Of File, and it returns a value of 1 if End of File is true, otherwise it returns a 0. EOF is useful when several files are LINKed together, and a FIND on one file causes implicit FINDs in the linked files (See LINK for more details). The RECORD function can be used in a similar way for the current file, it is 0 for end of file.

EXAMPLE:

```
NOTE files "mail_list" and "agent_file" both have
NOTE    key name
OPEN "mail_list" AS mail
OPEN "agent_file" AS agent
LINK FILE agent KEY name TO FILE mail mail.name
FIND FILE mail KEY name EXACT "Roger Moore"
IF EOF(agent) THEN
  PRINT "Roger Moore is not an agent."
ENDIF
```

will open the **mail_list** and **agent_file** files, link the agent file to the mail file, and do a find in the mail file for "Roger Moore". This will cause an implicit find in the agent file (because of the LINK statement). The **IF EOF(agent) THEN** statement tests to see if the person's name was not found in the agent file, and if so

Roger Moore is not an agent.

is PRINTed out.

ERROR

USAGE:

ERROR

ERROR is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the number of the error. ERROR is typically used inside an error trapping routine to test which error occurred and to allow the appropriate action to be taken. If no error occurred then ERROR returns 0. See the list under Error numbers in the appendix. RESUME, RESUME AT and RETRY reset ERROR to 0.

EXAMPLE:

```
SET TRAP TO trap
PRINT "enter a number: ";
LABEL enternum
INPUT n
PRINT 10/n
END

LABEL trap
IF ERROR=19 THEN
  PRINT "ERROR-division by zero"
  PRINT "re-enter"
  RESUME AT enternum
ELSE
  HELP ERROR
ENDIF
```

will prompt the user to enter a number. If a zero is entered (causing a division by zero error), then

```
ERROR-division by zero
re-enter
```

appears on the screen and the operator can then re-enter the number. An explanation of the error will appear if it is not a divide by zero error.

ESCAPE

USAGE:

ESCAPE

PURPOSE AND OPERATION:

Return the ASCII value of the key which was used to complete an ENTER statement. Pressing the **RETURN** or **ENTER** key will return a value of 13, pressing the **ESC**ape key will return a value of 27.

EXAMPLE:

```
ENTER name
IF ESCAPE=27 THEN
   GOTO done
ENDIF
  .
  .
  .
LABEL done
PRINT "Data entry complete"
END
```

EXECUTE

USAGE:

```
EXECUTE ($)
```

where $ is any TEXT value that contains an IMS command.

PURPOSE AND OPERATION:

To execute TEXT as if it were an IMS command. EXECUTE can not be given TEXT values which are module declarations, variable declarations or program control statements like GOTO, GOSUB, CHAIN, END, loops, or error trapping. It can be given other commands, including file commands, field assignments and CALL statements. If the execute statement is passed a null text value, "", then IMS will keep accepting text as commands until an END statement is typed or the ESCape key is pressed. In this way an interactive mode is possible from inside an IMS module.

EXAMPLE:

1.
```
EXECUTE ("LIST KEY NAME")
EXECUTE ("FIND FIRST")
EXECUTE ("PRINT NAME")
PRINT "Enter a command: ";
INPUT c$
EXECUTE c$
```

will list all the records by their NAME key, then find the first record and print the field called NAME. (This is assuming a file is OPEN and the current file has a key called NAME, and a field called NAME). Then it will prompt you for a command and attempt to execute it.

2.
```
PRINT "Enter IMS Commands, type END to stop"
EXECUTE("")
```

will output:

```
Enter IMS Commands, type END to stop
IMS:
```

then EXECUTE the TEXT values entered until the user types **END** or presses the **ESC**ape key.

EXIT

USAGE:

```
looptype
  ...
  EXIT
  ...
endlooptype
```

where **looptype** is one of LOOP, WHILE, REPEAT and **endlooptype** is one of ENDLOOP, ENDWHILE or UNTIL.

PURPOSE AND OPERATION:

To leave a loop. The EXIT statement is useful for situations in which a certain condition is detected that should stop execution of the loop. Execution will then continue on the next statement after the loop. EXIT statements may only be inside a loop, and there may be as many EXITs as necessary.

EXAMPLE:

```
count=1
WHILE count <= maximum DO
  IF subtotals(count) = amount THEN
    EXIT
  ENDIF
  count=count+1
ENDWHILE
```

These statements will search through the elements of an array called **subtotals** for a match with **amount**. If found it will EXIT from the WHILE loop with **count** pointing to that element.

USAGE:

Any place in the syntax requiring a value, an expression must appear. The implied data type of a function, or the more complex data type of the two operands of a binary operator, yield a value of that type. Expressions are in fact built up recursively. To find out all the functions and operators and allowable syntax for expressions, refer to the syntax summary in the appendix.

PURPOSE AND OPERATION:

To dynamically generate a value. An expression can fall into any of four groups:

1. **Numeric expression**

A numeric expression can be simply a number, like **5**, or a numeric variable, like **n**. It can also be several numbers or numeric variables combined using operators (see OPERATORS). Examples of this are **1+3**, **n/27+33**, and **n*n+37**. Numeric functions like **ABS(n+5)+33** and **MAX(33+22,ABS(2+5)+33,27)** are also expressions. Note that expressions can be inside expressions; the MAX function above shows this. To see how it works, we have to see the intermediate steps of evaluation:

```
MAX(33+22,ABS(2+5)+33,27)
```

equals

```
MAX(33+22,7+33,27)
```

equals

```
MAX(55,40,27)
```

equals

```
55
```

Therefore:

```
PRINT MAX(33+22,ABS(2+5)+33,27) + 2
```

would output:

```
57
```

In the above examples all the numbers and variables were assumed to be of type INTEGER, but REALs, LONGs, and INTEGERs can all be used together in a numeric expression. The resulting value is of the same type as the **highest** used in the expression. The order of types is as follows:

REAL : highest
LONG : second highest
INTEGER : lowest

Thus in an expression like **3.*2**, the answer is **6.**,

a REAL (a number containing a period is a REAL constant, see CONSTANT for more details). **3.** is a REAL, while **2** is an INTEGER, so the result is a REAL - the "higher" type. **2*2** equals **4**, since an INTEGER times an INTEGER returns an INTEGER. **37000/4** equals **9250**, a LONG, since a LONG - **37000**, with an INTEGER - **4**, gives the higher type - LONG.

In cases where the expression returns a type you don't want, the conversion functions REAL, LONG, and INTEGER can be used. For example, the expression **INTEGER(3.*2)** would return **6**, an integer.

2. **DATE expressions**

Dates are limited in how they can be used in expressions. Basically, adding and subtracting numeric expressions from dates and taking the difference in dates is supported. When a date is used in a numeric expression it is treated as a LONG. If a REAL is used in an expression with a DATE the expression will be type REAL, otherwise it will be type DATE. This is a problem in taking the difference in dates, because you want the number of days between the dates - not a DATE value returned. The answer is to apply the INTEGER function on the expression. For example, if **dt** and **dt2** are both date variables then

```
INTEGER (dt-dt2)
```

would be the expression for the number of days between the dates.

A TEXT expression can be assigned to a DATE variable (the TEXT is converted to DATE using the current SET DATE format).

3. **TEXT expressions**

A text expression can be a text constant, variable or function or several constants, variables or functions combined with the **+** (concatenation) operator. It can also be the null text value, **""**, which is 0 characters.

An example is:

```
LEFT$("Hello there, world!",5)
```

which returns "Hello".

TEXT expressions can also be inside TEXT expressions:

```
LEFT$(LEFT$("Hello there, world!",5),2)
```

equals

```
LEFT$("Hello",2)
```

equals

```
"He"
```

4. **Boolean expression, condition**

This type of expression is merely a form of numeric expression. As the name implies, it is used to represent a true or false value. Zero (0) is interpreted as FALSE, while any non-zero value is interpreted as TRUE. For example:

```
ABS(2-6)>3 AND 40>=LENGTH("Hello")
```

is FALSE (0); the steps are:

```
ABS(2-6)>3 AND 40>=LENGTH("Hello")
```

equals

```
-4>3 AND 40>=5
```

equals

```
0 AND 1
```

equals

```
FALSE AND TRUE
```

equals

```
FALSE
```

EXAMPLE:

```
INTEGER n
DATE dt,dt2
PRINT MAX(33+22,ABS(2+5)+33,27)
PRINT "Enter a number: ";
INPUT n
PRINT n/7+33
PRINT 2*3.
PRINT INTEGER(2*3.)
dt="June 21,1985"
PRINT dt+7
PRINT "Enter a date: ";
INPUT dt2
n=ABS(INTEGER(dt-dt2))
PRINT n
```

will output:

```
55
```

then prompt you to enter a number. After a number is entered, the value of the number divided by 7 plus 33 will be printed. Then:

```
6
6
June 28,1985
```

will be printed followed by a prompt to enter another date. After a date is entered, the number of days between the input date and June 21,1985 will be printed.

FIELD

USAGE:

```
FIELD(n)
   or
FIELD($)
```

where **n** is a number greater than zero, and **$** is a TEXT expression which evaluates to the name of a field.

FIELD is a function that returns the value of a field and may only be used in an expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return the value of a field. FIELD(**n**) returns the value of the nth field, so FIELD(1) would return the value of the first field. FIELD(**$**) where **$** is the name of a field, returns its value. Therefore if a file is OPEN with a field name of **NAME**, PRINT NAME or PRINT FIELD("NAME") will produce the same result. FIELD should only be used when a file is currently OPEN. It is useful in situations where the program asks the operator for the name of a field and then manipulates the field using the FIELD function.

EXAMPLE:

```
NOTE file "mail_list" has fields NAME, ADDRESS1,
NOTE  ADDRESS2, ADDRESS3, and PHONE in that order
OPEN "mail_list"
FIND FIRST
PRINT NAME,ADDRESS1,ADDRESS2,ADDRESS3,PHONE
PRINT FIELD(1),FIELD(2),FIELD(3),FIELD(4),FIELD(5)
PRINT "Name of a field: ";
INPUT a$
PRINT FIELD(a$)
```

will OPEN file "mail_list", find the FIRST record in this file, then PRINT out the five fields. Then it will repeat the printout of these fields, prompt the user to enter the name of a field, input the field name, and print the value of that field.

field name

USAGE:

filetag.field
or
.field
or
field

where **filetag** is the tag of the data file containing the field and **field** is the name of the desired field. If the file tag is missing and the first character of the field name is a period (.), the current file is assumed. If only the field name is specified, all open files will be searched for the field.

PURPOSE AND OPERATION:

To reference the location of a particular field in a file's data record. This referencing is treated uniformly with the referencing of user-declared variables, thus, specifying a field name will allow its value to be used in any expression. Additionally, if the reference is on the left side of an assignment statement, a new value may be assigned to it.

EXAMPLE:

```
OPEN "mail_list"
USE FILE mail_list KEY name
SCAN ALL FOR name>"M"  print name  LET code=""
```

will, for all people in file mail_list whose name is alphabetically greater than "M", print the name of that person and change their postal code to "".

FIND

USAGE:

FIND **file tag key clause mod**

where **file tag** and **key clause** are optional and **mod** is one of the following:

APPROX **value**
EXACT **value**
FIRST
LAST
NEXT
PREVIOUS
RECORD **numeric expression**

where **value** is an optional expression of the same type as the key. If no value is specified, the key expression is evaluated using the current field values.

PURPOSE AND OPERATION:

To find a particular record in a file. FIND will look in the current file if **file tag** and **key clause** are not present, otherwise the stated file and key will be searched. **APPROX** indicates an exact match or the next greater key if there is no exact match, **EXACT** an exact match, **FIRST** and **LAST** the first and last records, **NEXT** and **PREVIOUS** the next and previous records, and **RECORD** the record number. If none of these **mods** is present then APPROX is assumed. When FIND NEXT is used after the last record is found, RECORD becomes equal to zero (see RECORD), EOF (**file tag**) becomes equal to one, and the next FIND NEXT returns the first record of the file. A corresponding reverse action occurs with FIND PREVIOUS.

EXAMPLE:

```
NOTE NAME is a key in "mail_list"
OPEN "mail_list" AS MAIL

FIND Key NAME "John Smith"
PRINT NAME
FIND NEXT
PRINT NAME
FIND LAST
WHILE RECORD<>0 DO
  PRINT NAME
  FIND PREVIOUS
```

```
ENDWHILE
NAME="John Smith"
FIND
PRINT NAME
```

This will do an approximate FIND for "John Smith", PRINT the NAME, FIND the next record and PRINT the NAME. Then a backwards listing of the file is produced. The NAME field is then assigned the value "John Smith". This value is used when no expression follows FIND.

If "mail_list" has five names in it, --"Alice Bettens", "Dan Rather", "John Smith", "Larry Jones", "Marvin Peate" -- it will output:

```
John Smith
Larry Jones
Marvin Peate
Larry Jones
John Smith
Dan Rather
Alice Bettens
John Smith
```

file tag

USAGE:

FILE **identifying tag**
or
identifying tag

where **identifying tag** is the optional tag the data base was OPENed under or the file name (without sub-directories or extension). The first form is used embedded in a statement which expects other parameters, the second form in statement or function which requires no other parameters.

PURPOSE AND OPERATION:

In either form, this causes the operation in which the file tag is used to explicitly reference a particular file. In most cases, its use is optional, and if it is not used, the current file will be assumed to be the desired file. The second form of the file tag is used in such situations as field names and the EOF function.

When **identifying tag** is used in one to the FILE statements (not in a function), it changes the **current file** to the identified one.

EXAMPLES:

```
NOTE this shows the use of the first form
OPEN "mail_list"
OPEN "mail_list2"
LIST FILE mail_list ALL
NOTE this shows the use of the second form
LIST FILE mail_list2 NEXT 10 PRINT mail_list2.name
```

will open **mail_list** and **mail_list2.** It will then list all the records in **mail_list**, and then print the name field for each of the first ten records in **mail_list2.**

GETKEY

USAGE:

GETKEY

GETKEY is a function that returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To input a single character from the keyboard. GETKEY will scan the keyboard until the operator types a character. It will not output the character; a PRINT statement will have to be used if the operator wishes to see the character typed. GETKEY is useful in places where the operator is given several choices, and hitting a single key is the simplest way to choose one (See KEY PRESSED).

EXAMPLE:

```
 ...
PRINT "7. Second last choice"
PRINT "8. Last choice"
PRINT
PRINT "Enter your choice :";
response$=GETKEY
PRINT response$
```

will take the character the operator types, store it in **response$**, and output it to the screen.

GOSUB ... RETURN

USAGE:

```
GOSUB label
 ...
LABEL label
 ...
RETURN
```

PURPOSE AND OPERATION:

To call a subroutine inside the current module. The specified label must be present in the module. The action of GOSUB is to execute statements after LABEL **label** until a **RETURN** statement is encountered. Execution will then resume immediately after the GOSUB statement. GOSUB is useful when a part of the module is used many times.

EXAMPLE:

```
a=0
GOSUB printit
a=1
GOSUB printit
END

LABEL printit
PRINT "the value of 'a' is ";a
RETURN
```

will output:

```
the value of 'a' is 0
the value of 'a' is 1
```

GOTO

USAGE:

```
GOTO label
  ...
LABEL label
  ...
```

PURPOSE AND OPERATION:

To continue execution at the labeled location in the current module. GOTO differs from GOSUB in that no return address is saved. Using GOTOs generally makes modules difficult to understand and should be avoided whenever possible.

EXAMPLE:

```
PRINT "at first print"
GOTO third
PRINT "at second print"
LABEL third
PRINT "at third print"
```

will output:

```
at first print
at third print
```

HELP

USAGE:

HELP **error number**
HELP **command**

PURPOSE AND OPERATION:

To print a message explaining an error or giving advice on using COMMANDS.

EXAMPLE:

1.
```
IMS:HELP FIND
```

will explain how to use the FIND command.

2.
```
SET ERROR TRAP TO trap
a = a/0
END

LABEL trap
HELP ERROR
END
```

will output:

Divide by zero attempted.

identifier

USAGE:

MODULE **identifier**
REAL **identifier**
INTEGER **identifier**
TEXT **identifier**
DATE **identifier**
LABEL **identifier**
FILE **identifier**
KEY **identifier**

PURPOSE AND OPERATION:

To identify a module, a variable, a file, a field, or a label. Identifiers must start with a letter, but may contain letters, digits and underlines. The dollar sign may be used in variable and field identifiers. Spaces and other punctuation cannot be used in an identifier. Using dollar signs at the end of a TEXT variable name is a good way to differentiate it from a numeric variable. The underline is best used instead of a space to make the identifier more readable. Note, reserved words in IMS, like **IF, NEXT, PRINT** are not allowed to be identifiers of variables or fields.

The field identifier has some interesting variations. When a field is specified with no period in the identifier, all files, starting with the current file, may be searched for the field. If the first character of the identifier is a period, only the current file is searched. When the period is found inside the identifier, the part of the identifier before the period is used to identify the file the field is in and the part after the period is the field name.

EXAMPLE:

```
REAL n,amount
TEXT NAME$ of 30
INTEGER number_of_employees

 ...
LIST PRINT DATA.ACCOUNT
```

Here n and **amount** are REAL identifiers, **NAME$** a TEXT identifier, **number_of_employees** an INTEGER identifier, and **ACCOUNT** a field name of file **DATA.**

IF ... ELSE ... ENDIF

USAGE:

```
IF condition THEN                IF condition THEN
 ...                    or        ...
ELSE                             ENDIF
 ...
ENDIF
```

PURPOSE AND OPERATION:

To make a test and carry out actions based on the result of that test. **condition** is an expression that returns a true or false indication when evaluated. **amount= total, subtotal > discount**, are conditions because they are either true or false. (Note that a relational sign, =, <, >, >=, <=, or >, is used - see RELATIONALS). The IF statement evaluates the condition, and if the condition is true, it executes the statements after the IF up to the ELSE (if ELSE is present). If the condition is FALSE, and ELSE is present, the statements after ELSE are evaluated. ENDIF marks the end of the conditional statements.

EXAMPLE:

1.
```
IF total<0 THEN
  PRINT "error: deductions are greater than gross"
ELSE
  PRINT total
ENDIF
```

If **total** is less than zero it will output:

```
error: deductions are greater than gross
```

or if **total** is greater than or equal to zero it will output the value of **total**.

2.
```
IF balance>50000.00 THEN
  PRINT "Careful: higher tax bracket!"
ENDIF
```

If **balance** is more than fifty thousand this will output:

```
Careful: higher tax bracket!
```

but will do nothing if **balance** is less than or equal to fifty thousand.

INPUT

USAGE:

INPUT **identifier list**

where identifier list is a list of field or variable **identifier** separated with commas.

PURPOSE AND OPERATION:

To input a value into a field or variable. The field or variable must have been previously declared. If it is a numeric field or variable and a non-numeric value is entered, the operator will be asked to re-enter the value. If a TEXT value is entered that is longer than the variable size, the extra on the right is ignored. INPUT does not write anything to the screen, so usually a prompt message is printed on the screen to tell the operator what to type in.

The input is usually entered from the keyboard, but the SET statement can be used to enter data from other sources (see SET INPUT FROM). The ENTER or RETURN key must be pressed after the data for each variable or field is typed in.

EXAMPLE:

```
INTEGER num
TEXT in$ OF 20

PRINT "Please enter the number: ";
INPUT num
PRINT "Please enter the name: ";
INPUT in$
```

will prompt the operator to enter values into variables **num** and **in$**.

INSERT

USAGE:

INSERT **file tag**

where **file tag** is optional and refers to an already OPENed file.

PURPOSE AND OPERATION:

To add a record to the file. INSERT will add a record to the current file, (if **file tag** is not present), or to the file named in the **file tag.** Typically the fields of the record have just had information entered into them by INPUT or ENTER statements, then INSERT is used to add that record to the file. Indexes are also updated when a record is INSERTed.

EXAMPLE:

```
NOTE NAME and ADDRESS are field of "mail_list"
OPEN "mail_list" as MAIL
PRINT "Enter the name: ";
INPUT NAME
PRINT "Enter the address: ";
INPUT ADDRESS
INSERT
```

will insert one record into the file "mail_list".

INTEGER

USAGE:

INTEGER (**e**)

where **e** is any expression which can be converted to a value in the integer's range (-32768 to 32767).

INTEGER is a function that returns a number and may only be used in a numeric expression (see EXPRESSION).

PURPOSE AND OPERATION:

To take a value and convert it into an integer. INTEGER will take a value like "12.34" and return 12. No rounding is done on **e**. The value of **e** should not be outside the range of an INTEGER, otherwise an error will occur.

EXAMPLE:

```
PRINT INTEGER (-2343.67)
PRINT INTEGER (32.0+45.)
PRINT INTEGER (34.8-23.4)
PRINT INTEGER ("-123")
```

will output:

```
-2344
77
11
-123
```

KEY

USAGE:

KEY

PURPOSE AND OPERATION:

To return the current key value of the current record. KEY returns a value of the same type as the current key and can only be used in an expression. If NOKEY is currently selected or there is no current file, an error will occur. If RECORD is currently zero, KEY may have an undefined value.

EXAMPLE:

```
MODULE customer_report

TEXT namekey
REAL total,subtotal

total=0

OPEN 'transaction'
OPEN 'customer'

LINK FILE transaction KEY name TO FILE customer
                                        customer.name

FIND FILE customer KEY name FIRST
WHILE RECORD DO
   PRINT name
   USE FILE transaction
   namekey=KEY
   subtotal=0
   WHILE RECORD AND namekey=KEY DO
      PRINT ,transdate,transamount
      subtotal=subtotal+transamount
      FIND NEXT
   ENDWHILE
   PRINT "Total purchases: ";subtotal
   total=total+subtotal
   FIND FILE customer NEXT
ENDWHILE
PRINT "Total Sales: ";total
END
```

key clause

USAGE:

```
CHECK    file tag     key clause
CLEAR        "            "
COPY         "            "       ...
DELETE       "            "       ...
FIND         "            "       ...
LINK         "            "       ...
LIST         "            "       ...
MARK         "            "       ...
REINDEX      "            "
SCAN         "            "       ...
UNMARK       "            "       ...
USE          "            "
```

where ... denotes extra parts in the statement.

PURPOSE AND OPERATION:

To specify which key the file command will work on. The word **KEY** followed by the name of key explicitly states which key to use. The name is the same as the key name used when the file was created. **NOKEY** is used to indicate sequential order, ie. record #1, record #2, record #3, etc. The **key clause** can be omitted and the current or last used key will be used instead. If no key has yet been used and none is specified then the **NOKEY** key will be assumed.

EXAMPLE:

```
NOTE "invoice_list" has fields VENDOR_NUMBER,
NOTE   INVOICE_NUM, DATE, AMOUNT with
NOTE   key VENDOR_NUMBER
NOTE
OPEN "invoice_list" AS INV
LIST
LIST KEY VENDOR_NUMBER
LIST NOKEY
```

will list the records in the default/current key order, which in this case is **NOKEY**. Then it will be followed by a list in vendor_number order, and finally by a list in the order in which they were entered.

KEY PRESSED

USAGE:

KEY PRESSED

KEY PRESSED is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To test if a key has been pressed on the keyboard and not read. KEY PRESSED returns 1 if there is a key being pressed at the keyboard, or 0 if no key is being pressed. KEY PRESSED is useful when you want to be able to alter an executing operation. For example, suppose that in a report module you want to be able to stop printing at any time during the report. In this case the module could give a prompt telling you to press any key to suspend execution, giving you a chance to examine the report. If you want to know which key was pressed, a GETKEY statement will have to be used. An important thing to remember is that if KEY PRESSED is true, meaning a character was pressed, it will remain true until the character is read by a GETKEY or INPUT statement.

EXAMPLE:

```
NOTE file "maillist" has already been OPENed and
NOTE   has fields NAME, ADDRESS1, and ADDRESS2
PRINT "Press any key to suspend report generation"
WHILE NOT EOF(maillist) DO
  PRINT NAME
  PRINT ADDRESS1
  PRINT ADDRESS2
  IF KEY PRESSED THEN
    a$=GETKEY
    PRINT "Press any key to continue operation"
    WHILE KEY PRESSED = 0 DO
    ENDWHILE
    a$=GETKEY
  ENDIF
ENDWHILE
```

These statements will print three fields from the **maillist** file. Inside the WHILE ... ENDWHILE loop a test is made to see if a key was pressed; if so, the module waits until another key is pressed before continuing execution.

LABEL

USAGE:

LABEL **labelname**

PURPOSE AND OPERATION:

To mark a position in a module. Labels are used by GOTO, GOSUB, SET TRAP TO, SET HEADER TO, SET FOOTER TO and RESUME AT statements to indicate the position at which to continue execution. The label itself is not executed; it is only a marker.

EXAMPLE:

1.
```
...
LABEL here
...
```

The **LABEL here** statement will have no effect on execution.

2.
```
...
LABEL here
...
GOTO here
```

Now **LABEL here** is a target for the **GOTO here** statement.

LEFT$

USAGE:

```
LEFT$($,n)
```

where **$** is any TEXT expression, and **n** is any numeric expression.

LEFT$ is a function that returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the leftmost **n** characters of **$**. If the value of **n** is greater than the length of **$**, all of **$** is returned. If the value of **n** is less than or equal to zero, the null TEXT value ("") is returned.

EXAMPLE:

```
a$="this is great!!"
PRINT LEFT$(a$,4)
```

will output:

```
this
```

which is the four leftmost characters of **a$**.

LENGTH

USAGE:

```
LENGTH ($)
```

where $ is any TEXT expression.

LENGTH is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the length of a TEXT value. LENGTH will return a number indicating how many characters are contained in $. The null TEXT value ("") has length zero.

EXAMPLE:

```
a$="this is some text"
PRINT LENGTH(a$)
PRINT LENGTH("more text")
PRINT LENGTH(a$+"more text")
```

will output:

```
17
9
26
```

LET

USAGE:

LET **identifier = expression**
or
identifier = expression

where **identifier** is a field or variable identifier.

PURPOSE AND OPERATION:

To assign the value of **expression** to an **identifier. identifier** is a previously declared variable or a field name and **expression** can be any type of expression of the same type (numeric, text or date) as **identifier** (See EXPRESSION). Use of LET is optional.

EXAMPLE:

1. LET amount = 500.00

2. total = amount+(rate*principal)-deductions

LIBRARY$

USAGE:

```
LIBRARY$($)
```

where $ is any TEXT expression.

LIBRARY$ is a function that returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return a TEXT value in "library" form. LIBRARY$ will convert all lower case letters to their upper case equivalents, change all characters which are not letters or digits to spaces, change multiple spaces to a single space and remove all leading and trailing spaces.

EXAMPLE:

```
a$="   The  Blind .Mice.  *_!!   "
PRINT "****";LIBRARY$(a$);"****"
```

will output:

```
***THE BLIND MICE***
```

LINE NUMBER

USAGE:

LINE NUMBER

LINE NUMBER is a function that returns a number and may only be used in a numeric expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return the current line of the printer page. When the SET PRINT TO statement is used to set up output to an alternative device, every PRINT statement to this device automatically updates this built-in variable. LINE NUMBER can be useful in reports to see if there is enough room on the page for a certain part of the report. LINE NUMBER can not be assigned a value, and is set to 0 when an EJECT PAGE command is executed or a footer subroutine is executed automatically.

EXAMPLE:

```
MODULE do_report
 ...
IF LINE NUMBER < 40 THEN
  GOSUB totals
ENDIF
 ...
```

Here in the module **do_report**, LINE NUMBER is checked to see if the current line is less than 40, (ie. there are at least 20 free lines on a 66 line page with 6 line bottom margin), and if so a subroutine to print out the totals is called, otherwise it is not called.

LINK

USAGE:

LINK **file tag1 key clause** TO **file tag2 expression**

where **file tag1** and **key clause** are both optional and refer to an existing OPEN file and a key in that file.

PURPOSE AND OPERATION:

To link one file to another, so that changes in accessing one file cause corresponding changes to the other. LINK allows relational as well as network capability; the information of several files can be "linked" together so that the total information can be easily manipulated. LINK's function is to cause a search in **file tag1** by **key clause** for a match with **expression** whenever the record in **file tag2** changes.

A payroll program is a good example. A file of employee data would keep information about each employee -- number, name, address, Year to date totals, etc. Another file would contain the hours worked, each record would consist of an employee number, the hours worked, pay-rate, etc. The two files would have to be separate because the data are different in both, but the payroll program needs information from each. LINK provides an easy solution, simply LINK the two employee number fields together and a FIND for the employee number in one file automatically finds the matching record in the other file. More than 2 files may be LINKed together; the only limit is the memory available.

EXAMPLE:

```
NOTE file "employee_data" has fields NUMBER,
NOTE   ADDRESS, and TOTAL_EARNINGS with key NUMBER
NOTE file "hours" has fields NUMBER, HOURS
NOTE   RATE
OPEN "employee_data" AS EMPDATA
OPEN "hours" AS HOURS
LINK FILE EMPDATA KEY NUMBER TO FILE HOURS
    HOURS.NUMBER
FIND FILE HOURS FIRST
PRINT "The data in 'hours' file is:"
LIST FILE HOURS CURRENT
PRINT "The data in 'employee_data' file is:"
LIST FILE EMPDATA CURRENT
```

will output:

```
          The data in 'hours' file is:
```

list the first record in the **hours** file, output

```
          The data in 'employee_data' file is:
```

and list the record in the **employee_list** file with the same NUMBER field as in the **hours** file record (the current record).

LIST

USAGE:

```
LIST file tag   key clause   range
  or
LIST STRUCTURE
  or
LIST BASES
```

where FILE **file tag**, **key clause**, and **range** are all optional and refer to an existing key in an already opened file and **range** is the range specification (see KEY CLAUSE and RANGE).

PURPOSE AND OPERATION:

To list a range of records. LIST will go through the stated file by the specified key, (or through the current file and key if not specified), outputting each record that matches the **range** specifications. If **range** is not specified then all records are listed. LIST STRUCTURE gives a report on the structure of the current data file. LIST BASES gives a report on all the files, with an asterisk,"*", by the current file and key index.

NOTE: if there is a PRINT in the range, only the expressions following PRINT will print for each record.

EXAMPLE:

```
NOTE "mail_list2" has fields NAME, ADDRESS1,
NOTE   ADDRESS2, PHONE, and ZIP
OPEN "mail_list2" AS MAIL
LIST ALL FOR ADDRESS2="Mudville"AND LEFT$(NAME,6)=
     "Jones,"
LIST STRUCTURE
PRINT
PRINT
LIST BASES
CLOSE MAIL
```

will first list all records in file **mail_list2** that have city **Mudville** and last name **Jones**, then output a report with a structure similar to the following:

FIELD NAME	TYPE	OPTION	OFFSET	LENGTH
name	text	0	1	30
address1	text	0	31	50
address2	text	0	81	50

```
phone          text    0          131        10
zip            text    0          141        9
```

The **OFFSET** column lists how many bytes there are from the start of the record to the particular field.

followed by the bases:

```
MAIL
     *DATA:              Record Length=150
                         No of Records=264
      INDEXES:     NAME          TYPE          LENGTH
                   ----          ----          ------
                  *NAME          text          30
```

LOCATE

USAGE:

LOCATE **row,col**

where **row** and **col** are numeric expressions corresponding to the screen row and column, respectively.

PURPOSE AND OPERATION:

To place the screen cursor at specific screen coordinates. The coordinates range from (1,1) at the upper left hand corner to (24,80) at the lower right hand corner of the screen. LOCATE is useful in positioning PRINT messages at certain locations on the screen.

EXAMPLE:

```
LOCATE (10,40)
PRINT "ERROR: unknown account number"
```

will put the cursor on the tenth row and the fortieth column, and PRINT

```
ERROR: unknown account number
```

at that screen location.

LONG

USAGE:

LONG (**e**)

where **e** is any expression which can be converted to a value in the range of a LONG INTEGER (-2147483648 to 2147483647).

LONG is a function that returns a numeric value and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the value of **e** as a LONG. LONG takes the value of **e**, whether it is an expression made up of REALs, LONGs, INTEGERs or TEXT values and returns a LONG value of it. LONG will truncate the value given to it, so that if it receives a value of "12.34", it will return 12. If the value of **e** is outside the LONG integer's range, an error occurs.

EXAMPLE:

```
PRINT LONG(34345.+7564224)
PRINT LONG(-534.23+1023.43)
PRINT LONG("-342.333")
```

which will output:

```
7598569
489
-343
```

LOOP ... ENDLOOP

USAGE:

```
LOOP
 ...
ENDLOOP
```

PURPOSE AND OPERATION:

To make a simple loop. LOOP ... ENDLOOP enclose the statements that are executed over and over until an EXIT, QUIT or END statement is executed. Note that there is no built in condition for LOOP ... ENDLOOP, making it different from the REPEAT and UNTIL loops. EXIT, END or QUIT must be used to leave a loop or you may get what is called an "infinite loop".

EXAMPLE:

1.
```
LOOP
  PRINT "HELP!  I'm trapped in an infinite loop!"
ENDLOOP
```

will output:

```
HELP!  I'm trapped in an infinite loop!
```

forever.

2.
```
count=1
LOOP
  PRINT month(count)
  IF count=12 THEN
    EXIT
  ENDIF
  count=count+1
ENDLOOP
```

will output 12 values (the elements of the array called **month**). Note that if the IF ... EXIT statements were not in the example the loop would keep printing forever or until an error occurred.

MARK

USAGE:

MARK **file tag key clause range**

where **file tag, key clause** and **range** are all optional and refer to a key in an OPEN file and **range** is the range specification (see RANGE and KEY CLAUSE).

PURPOSE AND OPERATION:

To mark records for later deletion. MARK will go through the stated file by the stated key, (or through the current file and key if not stated), and mark each record that matches the **range** specification. If **range** is not specified then the current record is marked. The marked records will act as they did before -- nothing is changed except that the function MARKED will return a value of 1 (TRUE) when the current record is marked -- until a REINDEX statement is used to delete all marked records (see REINDEX and MARKED) or the record is UNMARKED. MARK is useful in cases where the operator indicates individual records to be deleted. Deleting them one at a time might be slower than simply marking them and REINDEXing them later.

EXAMPLE:

```
NOTE: customer_list has key field NAME
OPEN "customer_list" AS CUST

FIND FILE CUST FIRST
WHILE RECORD <>0 DO
  PRINT NAME
  PRINT "Delete this one? (Y/N)";
  INPUT response$
  IF CAP$(response$)="Y" THEN
    MARK
  ENDIF
  FIND NEXT
ENDWHILE
REINDEX
```

which will go through all the records of file **customer_list**, print the NAME field of each one, and ask the operator if the record is to be deleted. If the operator enters **Y** or **y** then the record is MARKed.

MARKED

USAGE:

MARKED

MARKED is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return a value indicating if the current record is MARKED or not. MARKED will return a zero if the record is not marked, and a one if it is. A record which is marked is simply a record which was included in a previous MARK statement (see MARK). MARKED should only be used when a file is OPEN.

EXAMPLE:

```
markcount=0
OPEN "employee_data" as EMP

FIND FIRST
WHILE RECORD<>0 DO
  IF MARKED=1 THEN
    markcount=markcount+1
  ENDIF
  FIND NEXT
ENDWHILE
PRINT markcount; "records marked"
```

will output a number showing how many marked records are in file **employee_data.**

MASK

USAGE:

e MASK **m**

where **e** is any numeric or TEXT expression, and **m** is a TEXT expression that contains a valid mask value.

MASK is a function that returns a TEXT value and may only be used in expressions (see EXPRESSION) and ENTER and DISPLAY statements.

PURPOSE AND OPERATION:

To format a value. MASK will take a TEXT or numeric value and return a TEXT value in the format specified with the **m** mask. See SET DATE TO for the format of a DATE mask. MASKs may have an initial "start character":

The start character:

- ! - must enter a value into this field. A simple carriage return will not be accepted.
- ? - optional field. The user may or may not enter a value into this field.

If the first character is neither of these then the field is assumed to be an optional field.

Mask control characters:

- * - any character is accepted, except for control codes.
- # - must enter a digit, 0 ... 9, only. If the field is of a numeric type then a leading negative sign is allowed.
- l - letters entered are shown in upper or lower case, as they were entered. Digits, punctuation, etc., are not allowed.
- L - letters entered are shown in upper case, even if a lower case letter was typed. Digits, punctuation, etc., are not allowed.
- A - upper case alphanumeric; meaning both digits and numbers are accepted, with lower case letters shown in upper case.
- a - mixed case alphanumeric; meaning both digits and numbers are accepted, with letters shown in upper or lower case, as they were entered.
- \ - force the next character; the next character will not be seen as a mask control character but rather as a character to be copied in place.

. - decimal point, for numeric values this sets the position of the decimal point in the resulting formatted TEXT value.

^ - "ghost" character, this will make the next character go in the mask if the preceding character in the value is not a space or a minus sign, otherwise a space will appear in the mask.

All other characters are copied into the mask and left in place.

Also important to know are the default masks. These are the masks that are used if no mask is specified in the field descriptor, and they vary according to the type of variable in use. The default masks are:

TEXT - any upper or lower case letters, digits, punctuation characters, etc., - anything except control codes (this is the * mask character). The length of the field will be the same as that specified in the **OF LENGTH**.

INTEGER - any digit or a leading negative sign. The length of the field will be 6 characters.

LONG - any digit or a leading negative sign. The length of the field will be 11 characters.

REAL - any digits, a sign, decimal point, an E (for the exponent). The length of the field is 20 characters.

EXAMPLE:

1. 313 MASK "!###"

The exclamation mark specifies a "must enter" value, and the digits specify a 3 digit number. Since a leading negative sign is allowed in a numeric value, - those with type INTEGER, LONG, or REAL, values like -32 and -7 would be accepted by the mask. Numeric values are also right justified when entered (like a calculator). This will return:

313

2. "4732122" MASK "###-####"

No starting ! or ? character means that this defaults to an optional value. The mask requires 3 digits and then 4 digits to be entered; the dash - character is not one of the mask control characters so it is shown in place in the value. The dash character does not need to be typed in. This will return:

473-2122

3. "v2a1b4" MASK "L#L-#L#"

The **L** mask control character specifies that a letter must be entered, and that letter will be converted to upper case when entered. This will return:

V2A-1B4

4. "e3d7" MASK "\L-AAAA"

This specifies a backslash character, - \, that will view the next character - **L**, as not being a mask control character. The dash is shown in place in the value. The **AAAA** characters specify 4 digits or letters, the letters shown in upper case. Examples of what could be returned from this mask are **L-2B74**, **L-4326**, **L-A55D**, **L-H23K**. This mask will return:

L-E3D7

5. 123456.24 MASK "###^,###^,###^,###.##

This kind of mask is suitable for printing out large numbers. The ^ makes the next character print out only if the previous character is not a space or minus sign. In this case, there are six digits before the decimal place, and the mask provides for twelve. If the mask did not have the ^, this would output:

, ,123,456.24

We can see that there are spaces in front of the first two commas, and these are the commas that we don't want to print out. So the ^ mask control character becomes a convenient method of printing a space instead of a comma in this case. This example would return:

123,456.24

MAX

USAGE:

MAX(**e1**,**e2**, ...)

where **e1**, **e2**, ... is a list of two or more expressions of the same type (numeric, TEXT or DATE).

MAX is a function that returns a value of the same type as its arguments (see EXPRESSION).

PURPOSE AND OPERATION:

To return the value of the largest item in a list. There must be at least two items in the list.

EXAMPLE:

```
a=5
b=6
c=2
PRINT MAX(4,b,2*c)
PRINT MAX("apples","oranges")
```

will output:

```
6
oranges
```

MFREE

USAGE:

MFREE

MFREE is a function which returns the amount of memory available for data.

PURPOSE AND OPERATION:

MFREE is useful for checking how much memory is available. A program may use it to decide whether it can carry out a certain operation. It should be noted that TEXT expressions and the DELETE, UPDATE and SCAN statements use extra memory when they are executing.

If you run out of memory, you can try invoking IMSI with more memory (for example: IMSI program #8k).

EXAMPLE:

```
IMS: PRINT MFREE;" bytes free"
```

is an interactive statement which will print something like

```
3024 bytes free
```

MID$

USAGE:

MID$($,n1,n2)

where $ is any TEXT expression, and **n1** and **n2** are any numeric expressions.

MID$ is a function that returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return a TEXT value of **n2** characters from $, starting at the **n1**th character. If **n1** is less than or equal to zero then it is seen as 1, and if **n1** is greater than the length of $ then MID$ returns the null TEXT value. If **n2** is zero or less a null TEXT value is returned, and if **n2** is greater than the length of $ then only $ from the **n1**th character is returned.

EXAMPLE:

```
a$="this is the text"
PRINT MID$(a$,6,2)
PRINT MID$(a$,9,3)
PRINT MID$(a$,-1,0)
PRINT MID$(a$,9,LENGTH(A$))
```

will output:

```
is
the

the text
```

MIN

USAGE:

MIN(**e1**,**e2**, ...)

where **e1**, **e2**, ... is a list of two or more expressions of the same type (numeric, TEXT or DATE).

MIN is a function that returns a value of the same type as its arguments, and may only be used in expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the value of the smallest item in a list. There must be at least two items in the list.

EXAMPLE:

```
a=4
b=5
c=3
PRINT MIN(a,b+c,10-b,2*a)
PRINT MIN("apples","oranges")
```

will output:

```
4
apples
```

MODULE

USAGE:

MODULE **modulename**
or
MODULE **modulename (paramlist)**

where **modulename** is the name of the module and **paramlist** is a list or parameter identifiers separated with commas.

PURPOSE AND OPERATION:

To mark the beginning of an IMS program module, giving it a name and identify the parameters being passed to the program. Every IMS program must start with the MODULE statement. The program following MODULE can use the parameters like user defined variables except that parameters inherit their type from the calling module.

EXAMPLE:

```
MODULE run_hanoi

INTEGER i

PRINT "Number of Disks? ";
INPUT i

PRINT "Moving ";i;" disks from A to B"
CALL hanoi(i,"A","B","C")

END



MODULE hanoi(num,from,to,other)

IF num=1 THEN
   PRINT from;to;' ';
ELSE
   CALL hanoi(num-1,from,other,to)
   PRINT from;to;' ';
   CALL hanoi(num-1,other,to,from)
ENDIF

END
```

NOT

USAGE:

NOT **(condition)**

where **condition** is any expression that evaluates to true or false. Examples are **amount<total**, **balance>deductions**, which all have relationals like <, >, =, <=, >=, **CT**, etc. in them (see RELATIONALS).

NOT is only used in places where a condition is allowed -- in IF, WHILE, and UNTIL statements, as well as in range specifications.

PURPOSE AND OPERATION:

To test for when the condition is not true. If the condition is false, NOT will return true, if the condition is true, NOT will return false. NOT can be used with ANDs, ORs, and XORs, to build up larger conditions made from smaller ones.

EXAMPLE:

```
IF NOT (seriesno="23A10737") THEN
  PRINT "widget B series"
ELSE
  PRINT "widget A series"
ENDIF
```

will output:

```
widget B series
```

if **seriesno** is not equal to "23A10737", otherwise

```
widget A series
```

will be PRINTed out.

NOTE

USAGE:

NOTE **text**

PURPOSE AND OPERATION:

To allow for the module writer to document the module's operations. The NOTE and the following text are all ignored by IMS. Putting NOTEs in the module will not make it run faster or better, but are a good idea since they tell the person reading the module what the statements are supposed to do. NOTEs clarifying modules, subroutines, and tricky statements are all good ideas.

EXAMPLE:

```
NOTE This module gives a report based on the
NOTE   vendor maintenance files.
MODULE report
  ...
```

OPEN

USAGE:

```
OPEN "pathlist" AS file tag
  or
OPEN "pathlist"
```

where **pathlist** is the name of a file on disk, and **file tag** is a file identifier.

PURPOSE AND OPERATION:

To open a file. OPEN tells IMS to work with the file named in **pathlist**, with the **file tag** being the name for the file in the module. If the AS clause is not present then the filename becomes the file tag. You must OPEN a file before you can work on it, and it is then referred to by its **file tag**. Each file that is OPENed should be CLOSEd when work on it finishes.

EXAMPLE:

```
OPEN "/d0/acct_data/vendor_list" AS VENDORS
```

will open file **/d0/acct_data/vendor_list**, and then have it referred to as VENDORS throughout the module's statements.

operators

USAGE:

n1 op n2
or
t1 + t2

where **n1** and **n2** are any numeric expressions, and **op** is one of the following operators:

*	multiplication
/	division
+	addition
-	subtraction
%	modulus or remainder, (only for integers)

and **t1** and **t2** are any TEXT expressions:

+	concatenation

PURPOSE AND OPERATION:

To provide calculation power to IMS expressions the basic four operators (*, /, + and -) are present, as well as modulus (%). Modulus works only for two integer values and returns the remainder after the division. This is necessary because division using integers returns an integer number, ie, the remainder is lost. For example, 7/4 equals 1 (because 7 and 4 are both integers) and 7 % 4 equals 3. The TEXT operator, +, appends the second TEXT expression to the first. The result is a TEXT value with a length equal to the lengths of the 2 TEXT expressions added together.

EXAMPLE:

```
INTEGER count,number
REAL amount,benefits,rate,deductions
TEXT first$,second$ of 255

amount=500.00
benefits=225.50
rate=0.80
deductions=235.37
count=10
number=2
first$="the first"
second$=" and the second"

PRINT (amount+benefits)*rate-deductions
```

```
          PRINT amount/rate
          PRINT (count+number)%5
          PRINT count/3
          PRINT first$+second$
          PRINT second$+first$
```

will output:

```
          345.03
          625
          2
          3
          the first and the second
           and the secondthefirst
```

OR

USAGE:

condition1 OR **condition2**

where **condition1** & **condition2** are both any expression that evaluates to true or false. Examples are **amount<total**, **balance>deductions**, which all have relationals like <, >, =, <=, >=, **CT**, etc. in them (see RELATIONALS).

OR is only to be used in between two conditions to make up one larger condition, and therefore can only be used in places where a condition is allowed -- in IF, WHILE, WHEN and UNTIL statements, or in a range specification.

PURPOSE AND OPERATION:

To test for cases when **condition1** or **condition2** is true, or both are true. In other words, when at least one of **condition1** and **condition2** is true. OR returns true in this case; if neither condition is true then OR returns false. ORs can be used with AND, NOT, and XORs to build larger conditions made up of smaller ones.

EXAMPLE:

```
IF empnumber>maximum OR empnumber<minimum THEN
  PRINT "error in empnumber"
ENDIF
```

if **empnumber** is greater than **maximum** or **empnumber** is less than **minimum**, or if both are true then it will output:

```
error in empnumber
```

PADCENTER$

USAGE:

```
PADCENTER$($,n)
```

where $ is any TEXT expression, and **n** is any numeric expression that has a value of zero or more.

PADCENTER$ is a function that returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To center text. PADCENTER$ will take $ and center it in a TEXT value of length **n**. If **n** is larger than the length of $ then spaces will "pad" the returned text on the left and right. If **n** is less than or equal to the length of $ then $ is returned.

EXAMPLE:

```
a$="This is text."
PRINT "***";PADCENTER$(a$,20);"***"
```

will output:

```
***    This is text.    ***
```

PADRIGHT$

USAGE:

```
PADRIGHT$($,n)
```

where $ is any TEXT expression and **n** is any numeric expression with a value equal to or greater than zero.

PADRIGHT$ is a function that returns a TEXT value and may only be used in a TEXT expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return a TEXT value right justified. PADRIGHT$ will return $ right justified in a TEXT value of length **n**. If **n** is less than or equal to the length of $ then $ is returned. If **n** is larger than the length of $ then spaces are "padded" onto the left to make a TEXT value of length **n**.

EXAMPLE:

```
a$="This is text."
PRINT "****";PADRIGHT$(a$,20);"****"
```

will output:

```
***        This is text.***
```

PAGE NUMBER

USAGE:

```
PAGE NUMBER
```

PAGE NUMBER is a variable that returns a number and may only be used in a numeric expression.

PURPOSE AND OPERATION:

To return the current page number when PRINTing to the alternative print path, the device or file stated in the SET PRINT TO statement. PAGE NUMBER can not be assigned in the usual way; instead use the SET PAGE NUMBER TO **n** statement. PAGE EJECTs and footer traps (see SET FOOTER TO) increment the PAGE NUMBER variable.

EXAMPLE:

```
SET HEADER TO head
SET PRINT TO "/p"
 ...

LABEL head
PRINT PAGE NUMBER
 ...
RETURN
```

This is part of a report, with the header subroutine set to **head**. In the head subroutine the PAGE NUMBER is PRINTed out.

PRINT

USAGE:

PRINT **expression list** **endmark**

where **expression list** is a list of any type of expressions separated with commas or semicolons, and **endmark** is one of the following:

; - semicolon
, - comma
<no mark>

PURPOSE AND OPERATION:

PRINT will output a list of values. When expressions are separated with a comma, enough spaces are printed so that the next expression will be printed at the next column that is divisible by 16. PRINT will normally print on your terminal screen but this can be changed with several SET statements. SET SCREEN OFF will disable PRINT from printing on the screen. SET SCREEN ON will enable it. SET PRINT TO "/p" will establish **/p** as an alternative PRINT device. If you then SET PRINT ON, PRINT will send output to **/p**. SET PRINT OFF will disable PRINT to **/p** but will keep the path to **/p** open. In multi-user situations you may want to close the printer path (SET PRINT TO ""), so other users may use the printer. Instead of a printer you can SET PRINT TO **filename**, so you can edit your output or print it later.

The **endmark** character is either a semicolon (;) meaning don't start a new line, or a comma (,) meaning move over to the next column divisible by 16, or nothing () meaning start a new line.

EXAMPLE:

```
MODULE printtest
INTEGER a
REAL p
TEXT a$ of 255

a=34
p=123.25
a$="the number is "

PRINT "This is one print line"
PRINT "John's name:"
PRINT 'The title was "Gone with the wind"'
PRINT "note that the semicolon stops a new line";
```

```
PRINT " same line"
PRINT a$;a;", the amount is $";p
PRINT "the figures for the last month are:"
PRINT 234.32,123.23,1234.23,10.2
```

will output:

```
This is one print line
John's name:
The title was "Gone with the wind"
note that the semicolon stops a new line same line
the number is 34, the amount is $123.25
the figures for the last month are:
234.32          123.23          1234.23          10.2
```

Note that to print a single quote in **John's**, double quotes are needed to surround the text, and double quotes in "Gone with the Wind" must be surrounded by single quotes. Also note that the fourth output line came from two statements (the semicolon does not print a new line). Finally, note that several items can be output by a single PRINT statement.

QUIT

USAGE:

```
QUIT
```

PURPOSE AND OPERATION:

To stop all execution and return to the operating system or executive. QUIT differs from END in that QUIT stops all execution of the IMS program, while END will stop execution or the current module and return to the calling module. QUIT is useful for dealing with fatal errors, errors which mean the module cannot continue.

EXAMPLE:

```
IF ERROR = 51
  PRINT "HARDWARE ERROR - CHECK EQUIPMENT"
  QUIT
ENDIF
```

This will stop all if error 51 were encountered.

range

USAGE:

COPY	**file tag**	**key clause**	...	**range**
DELETE	**file tag**	**key clause**	**range**	
LIST	**file tag**	**key clause**	**range**	
MARK	**file tag**	**key clause**	**range**	
SCAN	**file tag**	**key clause**	**range**	
UNMARK	**file tag**	**key clause**	**range**	

where **range** refers to the optional range specification.

PURPOSE AND OPERATION:

To specify which records the file command will operate on, and which action to take on a record which meets the range specification. The range specification can be explicitly stated with each of the file commands, or it can be omitted, in which case the default for the file command is assumed. The defaults are ALL the records for the COPY, LIST and SCAN. For DELETE, MARK, and UNMARK, the current record, CURRENT, is the default. The range specification has 2 parts, the actual range of records to work on, and then the action to take on a record which meets the specification. There can, in fact, be zero or more actions specified in a range.

1. The Range
   the various choices are:
   a) ALL
      all the records in the file.
   b) ALL FOR **condition**
      all the records that satisfy the condition. The condition may compare variables, fields, and constants, with any of the relationals.
   c) ALL WHILE **condition**
      starting at the first record, all records until the condition is false.
   d) PREVIOUS **n**
      the previous **n** records, including the current record, where **n** is a numeric expression.
   e) CURRENT
      the current record.
   f) NEXT **n**
      the next **n** records, including the current record, where **n** is a numeric expression.
   g) PREVIOUS **n** FOR **condition**
      the previous **n** records that satisfy the condition.

h) PREVIOUS **n** WHILE **condition**
the previous **n** records or until **condition** is false.

i) NEXT **n** FOR **condition**
the next **n** records that satisfy the condition.

j) NEXT **n** WHILE **condition**
the next **n** records or until **condition** is false.

2. The Action

a) LOWEST **expression** TO **variable**
store the lowest value of **expression** in **variable.**

b) HIGHEST **expression** TO **variable**
store the highest value of **expression** in **variable.**

c) COUNT TO **variable**
store the number of records in **variable.**

d) TOTAL **expression** TO **variable**
total all the values of **expression** and store result in **variable.**

e) PRINT ...
standard PRINT statement.

f) LET ...
standard LET statement with assignment to field variables. For the SCAN and COPY commands.

NOTE: LOWEST, HIGHEST, COUNT and TOTAL will leave the value or the variable unchanged if no records are selected. More than one action may appear in a statement.

EXAMPLE:

```
NOTE file "inventory_list" has fields NAME,
NOTE    PARTNO, AMOUNT, and COST

REAL totalems
LONG num_records

OPEN "inventory_list" AS INV
LIST ALL FOR LEFT$(PARTNO,3)="3A4" AND AMOUNT<100
     AND COST >50.00
MODIFY ALL FOR AMOUNT>100 AND COST>100.00
     LET COST=COST*0.90 PRINT "NAME: ";NAME
SCAN ALL FOR TOTAL AMOUNT TO totalems
     COUNT TO num_records  PRINT "Total is:";
     total_items  PRINT "Number of records are:";
     num_records
FIND FIRST
LIST CURRENT
LIST NEXT 3
```

will list all the inventory records in the "3A4" series with amount in stock less than 100 and cost more than 50 dollars. Then it will reduce by 10 percent the cost of all inventory

records with more than 100 in stock and a cost of more then 100 dollars, and print the name of each it changes. Then it will scan through all the records and print the total number of items in stock and how many records are in the file. Finally it will LIST the first record and then the first 3 records.

REAL

USAGE:

```
REAL (e)
```

where e is any expression.

REAL is a function that returns a number and may only be used in a numeric expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return a number as a REAL value. REAL will take any expression and convert it to a REAL.

An error will result if a TEXT expression cannot be converted into a REAL.

EXAMPLE:

```
PRINT REAL(2+232)
PRINT REAL(-5285)
PRINT REAL("284.78")
```

will output:

```
234
-5285
284.78
```

RECORD

USAGE:

RECORD

RECORD is a function that returns a LONG number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the record number of the current record. Each record is numbered sequentially in the file. RECORD will return a zero if a previous FIND statement did not find a record. RECORD should only be used when a file is OPEN.

EXAMPLE:

1.
```
OPEN "names" as file1
FIND EXACT "JOHN SMITH"
IF RECORD<>0 THEN
  PRINT "FOUND"
ELSE
  PRINT "NOT FOUND"
ENDIF
```

which will output:

```
FOUND
```

if a record for **JOHN SMITH** is present in file **names** and

```
NOT FOUND
```

if it isn't.

2.
```
OPEN "names" as file1
FIND FIRST
WHILE RECORD <>0 DO
  PRINT NAMEFILE.NAME
  FIND NEXT
ENDWHILE
```

which will output all the names in file **names**.

REDO

USAGE:

```
looptype
  ...
  REDO
  ...
endlooptype
```

where **looptype** is one of LOOP, WHILE, REPEAT and **endlooptype** is one of the matching ENDLOOP, ENDWHILE, UNTIL statements.

PURPOSE AND OPERATION:

To ignore the remainder of the loop and continue execution at the start of the loop. REDO is a simple way of avoiding execution of statements at the bottom of a loop.

EXAMPLE:

```
a=0
WHILE a<8 DO
  a=a+1
  IF a=5 THEN
    REDO
  ENDIF
  PRINT a
ENDWHILE
```

will output:

```
1
2
3
4
6
7
```

note that the 5 was not output because of the IF ... REDO statement.

REINDEX

USAGE:

REINDEX **file tag** **key clause**

where FILE **file tag** and **key clause** are both optional, and refer to an existing key in an already OPENed file.

PURPOSE AND OPERATION:

To re-index an existing file and remove marked records. REINDEX is necessary when CHECK indicates a BAD condition, or to delete all MARKed records. REINDEX will go through the data file, remove MARKed and DELETEd records and generate new indexes in place of the old ones. REINDEX will not erase or change any unMARKed data.

EXAMPLE:

```
OPEN "mail_list" as MAIL
FIND FIRST
MARK
REINDEX FILE MAIL
```

which will delete the first record and any other MARKed records.

relationals

USAGE:

e1 r e2

where **e1** and **e2** are expressions of the same type -- either TEXT, numeric or DATE, and **r** is one of the following relationals:

=	or **EQ**	equal to
>	or **GT**	greater than
<	or **LT**	less than
>=	or **GE**	greater than or equal to
<=	or **LE**	less than or equal to
<>	or **NE**	not equal to

or if **e1** and **e2** are both TEXT values then the following relationals can be used:

BW	begins with
CT	contains
SL	sounds like

PURPOSE AND OPERATION:

To test one value versus another. A relational will return true (1) or false (0) according to the comparison of one value to another. For numeric expressions the testing is straight forward: 3>4 returns false, 7 LE 10 returns true, etc.

For TEXT expressions the ordering is according to the "ASCII" sequence (see ASCII table in Appendix). "JONES" > "SMITH" returns false, and "JENKINS" LT "JENSEN" returns true. The ASCII sequence is similar to the alphabet's order, except that lower case letters are considered higher than upper case letters. So "smith" > "SMITH" returns true, "ZZZ" < "aaa" returns true, etc. The purely TEXT relationals are also straightforward, BW returns true if **e1** begins with **e2**, CT returns true if **e1** has the value of **e2** anywhere in it, and SL returns true if **e1** sounds like **e2** (only the text up to a punctuation mark is compared in SL).

EXAMPLE:

```
INTEGER a,b
REAL r1
TEXT a$,b$
```

```
a=4
b=6
r1=4.
a$="SMITH AND WESSON"
b$="SMITH"

IF a<b THEN
  PRINT a;" is less than ";b
ELSE
  PRINT a;" is not less than ";b
ENDIF
IF r1 GE b THEN
  PRINT r1;" is greater than or equal to ";b
ELSE
  PRINT r1;" is less than ";b
ENDIF
IF a$ > b$ THEN
  PRINT a$;" is greater than ";b$
ELSE
  PRINT a$;" is not greater than ";b$
ENDIF

IF "yes it is" CT "yes" THEN
  PRINT "true"
ELSE
  PRINT "false"
ENDIF

IF "SCHMIDT" SL "SMITH" THEN
  PRINT "SCHMIDT sounds like SMITH"
ELSE
  PRINT "SCHMIDT doesn't sound like SMITH"
ENDIF
```

which will output:

```
4 is less than 6
4 is less than 6
SMITH AND WESSON is greater than SMITH
true
SCHMIDT sounds like SMITH
```

REPEAT ... UNTIL

USAGE:

```
REPEAT
 ...
UNTIL condition
```

PURPOSE AND OPERATION:

To loop until a **condition** becomes true. REPEAT marks the start of the loop, and UNTIL **condition** marks the end of the loop. **condition** is any expression that evaluates to true or false. Examples are **amount<total** and **balance>deductions**, which have relationals like <, >, =, <=, >=, **CT**, etc. in them (see RELATIONALS). The loop will continue executing until the condition becomes true. This makes the REPEAT loop execute at least once.

EXAMPLE:

```
REPEAT
  PRINT "Press ESC to continue: ";
  a$=GETKEY
UNTIL a$=CHR$(27)
```

which waits for the ESCape key to be pressed before continuing.

RESUME

USAGE:

```
SET TRAP TO label
 ...
LABEL label
 ...
RESUME
```

PURPOSE AND OPERATION:

To continue execution after an error. Typically RESUME will be in the error trapping statements as pointed to by SET TRAP TO **label**. The action of RESUME is to continue execution at the statement after the error-causing statement. This differs from RETRY which re-executes the error-causing statement. RESUME is helpful in that the error can be handled in the error trapping statements and then RESUME is used to go back to the main part of the module.

EXAMPLE:

```
SET TRAP TO TRAP
 ...
PRINT 10/0
PRINT "back in main"
END

LABEL trap
PRINT "error encountered"
RESUME
```

since dividing 10 by 0 is an error, the following will be output:

```
error encountered
back in main
```

RESUME AT

USAGE:

```
SET TRAP TO trap label
 ...
LABEL resume label
 ...
LABEL trap label
 ...
RESUME AT resume label
```

PURPOSE AND OPERATION:

To resume execution at a specific statement after an error is encountered. Typically RESUME AT will be used in the error trap to resume execution at a specific label in the program.

EXAMPLE:

```
SET TRAP TO trap

LABEL enter
PRINT "ENTER A NUMBER";
INPUT num
PRINT 10/num
END

LABEL trap
PRINT "error encountered"
RESUME AT enter
```

which will prompt the operator to enter a number. If the user enters a zero,

```
error encountered
```

is output and the prompt is given again.

RETRY

USAGE:

```
SET TRAP TO label
 ...
LABEL label
 ...
RETRY
```

PURPOSE AND OPERATION:

To re-execute the error-causing line. RETRY will typically be part of an error trapping subroutine that is pointed to by SET TRAP TO **label**. RETRY will transfer execution back to the error-causing line. RETRY is useful for those kinds of errors which happen by operator carelessness and are easily rectified.

Some precautions need to be taken in the event that the error causing line continues to cause an error. You may need to keep track of how many times the same error occurs and take appropriate action.

EXAMPLE:

```
SET TRAP TO trap
 ...
LABEL trap
PRINT "insert disk into drive A"
RETRY
```

RIGHT$

USAGE:

RIGHT$($,n)

where $ is any TEXT expression and n is any numeric expression.

RIGHT$ is a function that returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return a TEXT value of the n rightmost characters of $. If n is larger than the length of $ then the entire $ is returned. If n is less than or equal to 0 then the null TEXT value (""), is returned.

EXAMPLE:

```
a$="this is the text"
PRINT RIGHT$(a$,4)
PRINT RIGHT$(a$,100)
```

will output:

```
text
this is the text
```

ROUND

USAGE:

ROUND (**n1**,**n2**)

where **n1** is any numeric expression and **n2** is a numeric expression that evaluates to a positive integer.

ROUND is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return a number to a given number of decimal places. ROUND will return the value of **n1** with **n2** decimal places. If **n1** has more than n2 decimal places then **n1** is "rounded up" at the **n2th** place. If **n2** is less than zero then it is taken to be zero.

EXAMPLE:

```
PRINT ROUND(1.556,2)
PRINT ROUND(-1.556,2)
PRINT ROUND(334.4567,0)
```

will output:

```
1.56
-1.56
334
```

SCAN

USAGE:

SCAN **file tag key clause range**

where **file tag, key clause,** and **range** are all optional and refer to an existing key in an already opened file and **range** refers to the range specification (see RANGE and KEY CLAUSE).

PURPOSE AND OPERATION:

To change a range of records. SCAN will go through the specified file by the specified key, (or through the current file and key if none is specified), and change fields in records that match the **range** specification. The changes are done through LET statements within the range specification, which assign the field(s) a new value. If there is no LET in the **range** then no update of the file takes place.

EXAMPLE:

```
NOTE  "inventory_list" has fields NAME, PARTNO,
NOTE  AMOUNT and COST
OPEN "inventory_list" AS INV
SCAN ALL FOR AMOUNT>100 LET COST=0.90*COST PRINT
     "name: ";NAME, "part number: ";partno
CLOSE FILE INV
```

will go through all records in file **inventory_list** and reduce the cost by 10 percent, of those items with more than 100 in stock. It will also PRINT the name and part number of each record it changes.

SET

USAGE:

SET **command**

where the possible SET **commands** are:

SET BOTTOM MARGIN TO **n**
SET DATE TO **format specs**
SET FOOTER TO **footer label**
SET FOOTER OFF
SET FORM TO **pathlist**
SET FORM OFF
SET HEADER TO **header label**
SET HEADER OFF
SET INPUT FROM **pathlist**
SET INPUT <ON|OFF>
SET LEFT MARGIN TO **n**
SET PAGE NUMBER TO **n**
SET PRINT TO **pathlist**
SET PRINT <ON|OFF>
SET RIGHT MARGIN TO **n**
SET SCREEN <ON|OFF>
SET SINGLE <ON|OFF>
SET TOP MARGIN TO **n**
SET TRAP TO **label**
SET TRAP OFF

where <ON|OFF> means that either ON or OFF should be stated, and **n** specifies a numeric expression.

PURPOSE AND OPERATION:

To change various IMS parameters. These include where the module sends output and gets input, turning screen output on or off, setting the DATE format, setting the printer parameters, setting the error traps, and other traps. The various commands are:

1. SET BOTTOM MARGIN TO **n**

   This sets the bottom margin on the printer page to start at the **n**th line. The default is 60 to leave a 6 line footer and margin at the end of a 66 line page (the footer starts at this line). SETting the bottom margin to 0 disables automatic footers (see SET FOOTER TO).

2. SET DATE TO **date format**

This sets the format in which DATE expressions are to be entered or output. **date format** is a text expression made up of several format control characters:

LETTER	EXPLANATION	EXAMPLE
Y	long year	1985
y	short year	85
M	long month	June
m	short month (3 characters)	Jun
N	long month digit	06
n	month digit	6
D	long day	02
d	day	2
W	long weekday name	Saturday
w	short weekday (3 characters)	Sat
****	next character is literal	

Other letters which are in the date format are left in place, including spaces. For June 2, 1985 the date format "y/n/d" would produce -- **85/6/2.** The default date format is "M d, Y" and produces a date in the form of **June 2, 1985.**

Checks are made to ensure that you have only one year, month and day specified in the date format.

3. SET FOOTER TO **footer label**

This sets the subroutine to call when the footer is to be printed on the report page. Footers are text that are printed at the bottom of the page. The footer subroutine is typically part of a generated IMS report module, and it is called when the LINE NUMBER variable (the number of lines printed on the page) is the same as the number in the SET BOTTOM MARGIN TO statement.

4. SET FOOTER OFF

This turns off the action caused by a previous SET FOOTER TO statement so that the footer subroutine is no longer called to print the footer.

5. SET FORM TO **pathlist**

pathlist is a text expression. This displays the form named on the screen, and causes subsequent DISPLAYs and ENTERs to be done using that form. It will not display the field information on the form.
NOTE: do not use the file extension for the form name.

6. SET FORM OFF

This releases the memory reserved for the form. Subsequent DISPLAYs and ENTERs may not be used before another SET FORM TO statement is executed.

7. SET HEADER TO **header label**

This sets the subroutine to call when the header (the title) is to be printed at the top of the report page. The header subroutine is typically part of a generated IMS report module, and is called when the LINE NUMBER variable (the number of lines printed on the page) is the same as the number in the SET TOP MARGIN TO statement.

8. SET HEADER OFF

This turns off the action caused by a previous SET HEADER TO statement so that the header subroutine is no longer called to print the header.

9. SET INPUT FROM **pathlist**

pathlist is a text expression and names the file or device to receive input from when SET INPUT ON is in effect. If **pathlist** is NULL ("") then the previous path is closed.

10. SET INPUT <ON|OFF>

If ON then input comes from the device or file named in the previous SET INPUT FROM **pathlist** statement, if OFF then input comes from the terminal keyboard. When the end of the alternative input path is reached, INPUT reverts back to the keyboard.

11. SET LEFT MARGIN TO **n**

This sets the left margin on the printer page to the **n**th column. The default is column 1.

12. SET PAGE NUMBER TO **n**

This sets the page counting variable, called PAGE NUMBER to **n**.

13. SET PRINT TO **pathlist**

pathlist is a text expression and names the file or device to send output to when SET PRINT ON is in effect. Footers, headers, margins, EJECT PAGE, PAGE NUMBER and LINE NUMBER relate only to this file or device, also called the **alternative print path**. If **pathlist** is NULL ("") then the previous path is closed.

14. SET PRINT <ON|OFF>

If ON then output is sent to the device or file named in the previous SET PRINT TO **pathlist** statement. Output is always sent to the screen unless SET SCREEN OFF is in effect.

15. SET RIGHT MARGIN TO **n**

This sets the right margin on the printer page to the **n**th column. The default is column 80.

16. SET SCREEN <ON|OFF>
This turns screen output on or off.

17. SET SINGLE <ON|OFF>
This SET statement is to control continuous versus single sheet printing. SET SINGLE ON means that after every page printed, IMS will give the message on the screen:

Insert new page and press RETURN to continue:

and wait for you to put another sheet in the printer before continuing.

18. SET TOP MARGIN TO **n**
This sets the top margin on the printer page to **n** lines. The default is 1 line. SETting the top margin to 0 disables automatic headers (see SET HEADER TO).

19. SET TRAP TO **label**
This sets the position to go to when errors occur during execution of the module. SET TRAP TO should be near the front of the module. Its action is to tell IMS to go to position **label** if any error occurs. The statements after the label form an error trap, and statements RESUME, RETRY, and RESUME AT help handle the error. The last SET TRAP TO statement executed is the one currently in effect. See the appendix for ERROR NUMBERS to see what errors are possible.

20. SET TRAP OFF
This turns off the error trapping set by SET TRAP TO. When the error trap is OFF then errors cause an error message to be displayed and halt the program.

EXAMPLE:

1.
```
SET PRINT TO "/p"
SET PRINT ON
SET SCREEN OFF

SET TOP MARGIN TO 2
SET BOTTOM MARGIN TO 62
SET LEFT MARGIN TO 1
SET RIGHT MARGIN TO 80
SET HEADER TO top
SET FOOTER TO bot
SET SINGLE OFF
SET INPUT FROM "/d0/data"
SET INPUT ON

 ...
LABEL top
```

```
  ...
RETURN
LABEL bot
  ...
RETURN
```

This will set output to the printer and turn screen output off. Then it will set the page margins, the headers, the footers, and the single sheet printing mode off. Input will come from file **/d0/data.**

Note: The printer may miss a character when printer and screen output are both on, and **tmode pause** is on.

2.
```
SET TRAP TO trap
SET FORM TO "wagesform"
  ...
ENTER number
PRINT 10/number
  ...
END

LABEL trap
PRINT "error encountered"
RESUME
```

which will show the **wagesform** form on the screen, then enter a field called **number** from the form. If the operator enters 0, causing a divide by zero error, execution continues at the label **trap,**

```
error encountered
```

will be printed and RESUME will continue execution after the error.

3.
```
DATE dt
dt="June 23,1985"
SET DATE TO "W M d, Y"
PRINT dt
SET DATE TO "Y M d"
PRINT dt
SET DATE TO "y-n-d"
PRINT dt
```

which will output:

```
Sunday June 23, 1985
1985 June 23
85-6-23
```

SHELL

USAGE:

```
SHELL $
```

where $ is a TEXT expression.

PURPOSE AND OPERATION:

To send the operating system a command. The text expression should have a length no greater than 80 characters. For valid commands see your operating system reference manual.

EXAMPLE:

```
SHELL "dir"
```

will give the directory.

SIGN

USAGE:

SIGN (**n**)

where **n** is any numeric expression.

SIGN is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the sign of a numeric expression. SIGN will return -1 if **n** is negative, 0 if **n** is equal to zero, and 1 if **n** is positive. SIGN is useful when the sign rather than the magnitude of the number is important.

EXAMPLE:

```
a=-2.3
PRINT SIGN(a)
PRINT SIGN(4)
PRINT SIGN(0)
```

will output:

```
-1
1
0
```

SOUND$

USAGE:

```
SOUND$($)
```

where $ is a TEXT expression.

SOUND$ is a function that returns a text value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return a text value indicating the sound of $. SOUND$ is useful for applications where the exact spelling of a name is not known, but the sound of it is. The conversion or text to sound stops at the first non-alphabetic character. The sound is stored in a coded form in a maximum of four characters.

EXAMPLE:

```
NOTE file "mail" has keys NAME and NAMESND and
NOTE field NAME
PRINT "Name to search for: "; OPEN "mail"
INPUT in_name$
FIND KEY NAMESND EXACT SOUND$(in_name$)
LIST NEXT 100 WHILE name SL in_name$
```

This will prompt and input a name value, then search an index of the sounds of the name field for the sound of the name you entered. Up to 100 records which sound like (SL) the name will then be listed.

SQRT

USAGE:

SQRT(**n**)

where **n** is a numeric expression.

SQRT is a function that returns a number and may only be used in a numeric expression (see EXPRESSION).

PURPOSE AND OPERATION:

To return the square root of **n**. If n is negative then an error is generated.

EXAMPLE:

```
PRINT SQRT(9)
```

will output:

```
3
```

SUBSTR

USAGE:

SUBSTR (**t1**,**t2**)

where **t1** and **t2** are TEXT expressions.

SUBSTR is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To find if and where a TEXT value is inside another TEXT value. SUBSTR returns a number indicating at which character the **t1** text starts in **t2**, and a zero if it is not present. SUBSTR is useful in cases where the text may have many characters, but only a specific TEXT value is important in the text.

EXAMPLE:

```
PRINT "enter your answer: ";
INPUT response$
IF SUBSTR ("YES",CAP$(RESPONSE$)) <>0 THEN
  PRINT "affirmative action taken"
  ...
ENDIF
```

will output:

affirmative action taken

if the user's response had YES (case ignored) in it.

TAB

USAGE:

PRINT TAB(**n**)

where **n** is any numeric expression with a value greater than zero.

TAB is a function that can only be used in a PRINT statement.

PURPOSE AND OPERATION:

To move the screen cursor or printer printhead to the **n**th column. The columns are numbered starting at 1 in the left margin, and increasing to 80 or beyond at the right margin. TAB is a convenient way to output at a certain position on the screen or printer. If **n** is less than the column that the cursor is currently at, the tab is ignored. Another point to remember is that the screen output and the alternative print path named in the SET PRINT TO have separate TAB columns that are separately "remembered".

EXAMPLE:

1. PRINT TAB(10);"NAME";TAB(20);"NUMBER"

will output:

```
          NAME      NUMBER
```

note that "NAME" is on tab 10 and "NUMBER" is on tab 20.

TEXT

USAGE:

```
TEXT(e)
```

where e is any expression.

TEXT is a function that returns a value of type TEXT and may only be used in expressions of type TEXT (see EXPRESSION).

PURPOSE AND OPERATION:

To convert any value into a value of type TEXT.

EXAMPLE:

```
INTEGER group, part
TEXT inventory_item OF LENGTH 10
group=224
part=101
inventory_item=TEXT(group)+"-"+TEXT(part)
PRINT "inventory item: ";inventory_item
```

will output:

```
inventory item: 224-101
```

TIME

USAGE:

```
TIME
```

TIME is a function that returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the current time. TIME will return the current hours, minutes and seconds.

EXAMPLE:

```
PRINT TODAY;"  ";TIME
```

which will output the current date and time, for example:

```
JUNE 23,1985  12:10:56
```

TODAY

USAGE:

TODAY

TODAY is a function that returns a date and may only be used in DATE expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return the current date. TODAY returns the current year, month and day, and can be PRINTed out, assigned to date variable, or used in a condition.

EXAMPLE:

```
DATE dt

PRINT TODAY
dt=TODAY-7
PRINT dt
```

will output something like:

```
June 23,1985
June 16,1985
```

TRIM$

USAGE:

```
TRIM$($)
```

where $ is any TEXT expression.

TRIM$ is a function which returns a TEXT value and may only be used in TEXT expressions (see EXPRESSION).

PURPOSE AND OPERATION:

TO return $ without the leading and trailing spaces. If $ was only a TEXT value of spaces then the null TEXT value (""), is returned. TRIM$ is useful when the text may have starting and ending spaces, and these spaces are to be ignored. TRIM$ may also shorten TEXT values, thus requiring less storage.

EXAMPLE:

```
a$="   note starting and ending spaces    "
PRINT "****"; TRIM$(a$); "****"
```

will output:

```
***note starting and ending spaces***
```

TRUNCATE

USAGE:

TRUNCATE (**n1**,**n2**)

where **n1** and **n2** are numeric expressions.

TRUNCATE is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

To return a number with a given number of decimal places. TRUNCATE will return the value of **n1** with **n2** decimal places. If **n1** has more than **n2** decimal places then the extra places are dropped. If **n2** is less than zero then it is taken to be zero.

EXAMPLE:

```
PRINT TRUNCATE(1.556,2)
PRINT TRUNCATE(-1.556,2)
PRINT TRUNCATE(334.588,0)
```

will output:

```
1.55
-1.55
334
```

UNLINK

USAGE:

UNLINK **file tag**

where **file tag** is optional, and refers to an OPENed file used in a previous LINK statement.

PURPOSE AND OPERATION:

To unlink a previously linked file. UNLINK will deallocate the memory set up for the linking process, and allow different LINKs to be set up. The **file tag** listed in the UNLINK command must be the same as the second file tag in a previous LINK statement.

EXAMPLE:

```
NOTE file "mail_customer" has fields NAME, NUMBER,
NOTE   and ADDRESS and file "catalog" has fields
NOTE   NUMBER and COMMENT with KEY NUMBER
NOTE
OPEN "mail_customer" AS MAIL
OPEN "catalog" AS CAT
LINK FILE MAIL KEY NUMBER TO FILE CAT CAT.NUMBER
 ...
UNLINK FILE CAT
```

UNMARK

USAGE:

UNMARK **file tag** **key clause** **range**

where **file tag, key clause** and **range** are optional and refer to an existing key in an already OPENed file, and **range** refers to the range specification (see RANGE and KEY CLAUSE).

PURPOSE AND OPERATION:

To unmark previously MARKed records. UNMARK will go through the stated file by the stated key (or through the current file and key if they are not stated), and unmark each record that matches the **range** specifications. If **range** is not specified then only the current record is UNMARKed. These unmarked records will act as they did before, and will not be deleted when the REINDEX statement is executed. UNMARK is useful to correct a mistaken MARK statement.

EXAMPLE:

```
NOTE file "customer_list" has key field NAME
OPEN "customer_list" AS CUST

FIND FILE CUST KEY NAME FIRST
WHILE RECORD<> 0 DO
  IF MARKED THEN
    PRINT NAME
    PRINT "UNMARK THIS ONE ? (Y/N)";
    INPUT RESPONSE$
    IF CAP$(RESPONSE$)="Y" THEN
      UNMARK
    ENDIF
  ENDIF
  FIND NEXT
ENDWHILE
```

which will go through all the records of file **customer_list**, print the NAME field of all the MARKED records, and ask the operator if the MARKED record should be UNMARKed.

UPDATE

USAGE:

UPDATE **file tag**

where **file tag** is optional and refers to an open file.

PURPOSE AND OPERATION:

To change an existing record. UPDATE will change the current record in the stated file, (or in the current file if **file tag** is not present). Typically a specitic record has been found using FIND, the fields of the record then have information entered into them by INPUT or ENTER, and UPDATE is used to change the record.

EXAMPLE:

```
NOTE: NAME, ADDRESS are field names of
NOTE   file "mail_list"
OPEN "mail_list" AS MAIL
FIND EXACT "JOHN SMITH"
IF RECORD<>0 THEN
  PRINT "new name? ";
  INPUT NAME
  UPDATE
ENDIF
```

which will search for the record with name "JOHN SMITH" in file **mail_list**. If it is found, a new name is input and tne record updated.

USE

USAGE:

USE **file tag** **key clause**

where either **file tag** or **key clause** may be omitted, and **file tag** is the tag for a previously opened file, and **key clause** is a key in that file.

PURPOSE AND OPERATION:

To explicitly set the current file and key. USE will notify IMS that in the subsequent file operations, the specified file and/or key are to be used if no file or key clauses are given in the operation(s). If a file tag or key clause are stated in the subsequent file operations, they then become the new current file and/or key.

EXAMPLE:

```
NOTE "mail_list" has fields NAME, ADDRESS, and ZIP
NOTE   and NAME and ZIP are keys.
OPEN "mail_list" as MAIL
USE KEY ZIP
LIST
  ...
```

will open file **mail_list** and then list its contents by order of the ZIP key.

VALUE

USAGE:

VALUE(e)

where e is any expression.

VALUE is a function that returns a number and may only be used in numeric expressions (see EXPRESSION).

PURPOSE AND OPERATION:

Identical to REAL (see REAL).

WHILE ... ENDWHILE

USAGE:

```
WHILE condition DO
  ...
ENDWHILE
```

PURPOSE AND OPERATION:

To loop while a condition remains true. WHILE marks the start of the loop, and ENDWHILE marks the end of the loop. **condition** is any expression that evaluates to true or false. Examples are **amount<total** and **balance>deductions** which have relationals like **<**, **>**, **=**, **<=**, **>=**, **CT**, etc. in them (see RELATIONALS). The loop will continue to be executed while the condition remains true. Note the condition is at the start of the loop, not at the end like the REPEAT loop. This means that the loop will never be executed if the condition is false at the very start.

EXAMPLE:

1.
```
a=10
WHILE a<10 DO
  PRINT a
  a=a+1
ENDWHILE
```

will output nothing since the condition "a<10" was false to begin with

2.
```
a=5
WHILE a<10 DO
  PRINT a
  a=a+1
ENDWHILE
```

will output:

```
5
6
7
8
9
```

XOR

USAGE:

condition1 XOR **condition2**

where **condition1** and **condition2** are boolean expressions (see EXPRESSION and RELATIONALS).

XOR is only used between two conditions in order to build a more complex condition.

PURPOSE AND OPERATION:

To test for cases when either condition1 or condition2 are true, but not both. XOR will return true if only one of the stated conditions is true, and false otherwise. It differs from OR in that OR is true if both conditions are true.

EXAMPLE:

```
IF amount=subtotal1 XOR amount=subtotal2 THEN
  PRINT "Enter the TD1 claim amount: ";
  INPUT TD1
ENDIF
```

will prompt the operator to enter the TD1 amount if amount is equal to subtotal1 or amount is equal to subtotal2, but not if both conditions are true.

Index

D

E

F

G

S

INFORMATION MANAGEMENT SYSTEM

Clearbrook Software Group

APPENDICES

CLEARBROOK SOFTWARE GROUP
INFORMATION MANAGEMENT SYSTEM

APPENDICES

Release B
January 1, 1986

APPENDICES

APPENDIX A - INSTALLATION

There are two parts to installation of the IMS programs. Part one is to copy the programs to the hard disk or backup diskettes. Part two is to set up the universal terminal driver (UTD) for the terminals in your system.

To install the programs on your system disk, place original disk in a secondary disk drive. If your system disk is **/D0** and secondary drive is **/D2** type the OS9 command:

/d2/install /d2 /d0

Thus, the general syntax of the command is:

<original disk>/install <original disk> <system disk>

If your original has come on several disks, this same process will need to be repeated for each disk.

The install program will copy the following programs to the **CMDS** directory of your system disk:

ims	the executive (menu)
imsI	the interpreter
imsC	the compiler
imsD	the file creator
imsF	the screen form editor
imsR	the report generator
tx	the text editor
mkterm	terminal driver editor
assoc	associate terminal with driver
nmall	list defined terminals and drivers
tname	list driver associated with terminal
imsErrs	error message file for imsI, imsC, imsD
help.FE	help messages for imsF
help.RE	help messages for imsR
help.TX	help messages for tx

It will also create the directory **IMS** on your system disk. Tutorial and example files will be copied into this directory.

The directories **UTD** and **UTD/UTD_DRIVER_FILES** will be created in the **CMDS** directory.

Now that the programs have been copied, you need to configure the **UTD** in order for the software to run correctly. Refer to **appendix B** for explanation of how to configure the **UTD** and for an explanation of what it is and does.

If **IMS** is being installed on a non-standard system (where the CMDS directory is not in the root directory of the system disk) then one of the following solutions can be used:

1. Use a variation of the install command where instead of specifying the system device, specify the directory containing the CMDS directory.

EXAMPLE: if the CMDS directory is in /D0/USERS, use:

/d2/install /d2 /d0/users

This will create the **IMS** directory in the **/D0/USERS** directory.

2. To install the commands in a directory other than CMDS you must copy the files manually or modify a file named pathfile on the original disk. The file consists of a list of path data, one piece of data per text line. You must change the occurrences of CMDS to whatever directory you are using. The data is in the format:

subpath1 subpath2
or
subpath

If there are two subpaths, <original>/subpath1 is copied to <system>/subpath2. If there is only one subpath, <original>/subpath is copied to <system>/subpath.

APPENDIX B - UNIVERSAL TERMINAL DRIVER

The universal terminal driver (UTD) is a collection of programs which allow the user to maintain terminal functions in a transparent, terminal independent manner. This is done through the use of a table of values which may be set up for any make of terminal desired. In addition, this table of values, called a terminal driver, is then associated with a physical device name on the user's system.

The files used by the UTD utility programs and other programs using terminal independence reside in subdirectories of the current execution directory. The UTD directory is located in the CMDS directory and contains a file for each terminal installed in the system (term, t1, t2, etc.). It also contains the directory (UTD_DRIVER_FILES) which contain the driver files for the different types of terminal you are using (tvi910, qvt102, vt100, etc.).

IT IS IMPORTANT WHEN A UTD PROGRAM OR OTHER IMS PROGRAM IS INVOKED THAT THE CURRENT EXECUTION DIRECTORY CONTAIN THE UTD DIRECTORY.

The four UTD utilities are:

1. **mkterm** - make terminal driver. This program allows the user to create or alter any terminal driver. The command line syntax is:

```
mkterm term1 term2
  or
mkterm terminal
  or
mkterm
```

where **term1** is the name of an existing terminal driver, **term2** is the name for a new driver. **terminal** is an existing or new terminal driver. Terminal drivers are located in the UTD/UTD_DRIVER_FILES subdirectory of the current execution directory.
A terminal driver is a data file which contains information about the codes to send to a particular make and model of terminal to perform:

- cursor addressing
- clear screen and home cursor
- clear to end of line
- move cursor up, down, right and left
- video attributes
- line drawing characters
- etc.

EXAMPLE: for a QUME QVT102 terminal

mkterm qvtl02

This will create a new terminal driver called **qvtl02** or edit an existing driver of the same name. Several menus allow you to enter and modify data.

2. **assoc** - associate a terminal driver to a physical device. The command line syntax is:

assoc device driver
or
assoc device

If no terminal driver is specified, the specified physical device is removed from the list of known driver-device associations.

EXAMPLE: /term is connected to a QUME QVTl02

assoc term qvtl02

This will copy UTD/UTD_DRIVER_FILES/qvtl02 to UTD/term.

3. **nmall** - name all defined terminal drivers and all known physical device associations. Command line syntax:

nmall

EXAMPLE:

nmall

will output:

```
Universal Terminal Driver defined devices
------------------------------------------
     term

Universal Terminal Driver defined driver files
----------------------------------------------
     qvtl02
```

4. **tname** - name the terminal driver a physical device is associated with. Command line syntax:

tname device(s)

where **device(s)** means zero or more physical devices may be specified.

EXAMPLE:

tname term tl

will output:

```
term is associated with qvtl02
tl is not defined
```

The last three programs, **assoc**, **nmall** and **tname**, are fairly simply to operate with all processing done through the use of command line arguments. The **mkterm** program, however, requires some explanation.

The purpose of **mkterm** is to create or edit any desired terminal driver. Since a terminal driver is a table of values, strings primarily, the main action done by mkterm is the entering and displaying of these values. A menu allows you to select which terminal function to change.

For entry of number of rows and number of columns, the user selects the appropriate menu choices. At the prompt, the new number should be entered as a base 10 integer.

For entry of strings, including part of cursor addressing entry, the user again selects the desired terminal function. At the prompt, the string of ASCII characters constituting the sequence necessary to perform that function on the target terminal should be entered. In order to allow the user to enter control characters and other non-printable characters, there are a three mechanisms.

The first is the use of a caret (^). The caret placed in front of a character causes the control character of that character to be used. For example, ^m is seen as the control character CARRIAGE RETURN. If the controlled character is invalid, for example ^., the caret is ignored, yielding ".".

The second is the use of a dollar sign ($). Using a dollar sign allows the user to specify a character as a hexadecimal constant. The expected format for this is $XX, where X is assumed to be a hexadecimal character. Thus, $0d would be seen as CARRIAGE RETURN, and $41 would be seen as A. If X is not hexadecimal, it is assumed to be zero.

The third is the use of a backslash (\). The backslash placed in front of a character causes that

character to be viewed literally. I.E. - any significance that character has to mkterm is lost. Thus, to have a caret, dollar sign or backslash in the string, the user would specify \^, \$ and \\ respectively.

Finally, there is the specification of cursor addressing. This is the most complex function, requiring three strings, as described above, and two coordinate specifications. A coordinate consists of: row or column selection (**S**), cursor addressing type (**T**), and coordinate offset (**O**) - a value added to row or column before it is sent to the terminal. It should be noted here that the UTD assumes that the top left corner of the screen is addressed as 0,0. The user gives this coordinate specification in the form of:

S;T;O

S may be **R** for row or **C** for column. **T** is a base 10 integer in the range of 1 to 15. **O** is a base 10 integer in the range of 0 to 255.

The various addressing types supported by the UTD are:

1 - coordinate is transformed linearly. I.E. - any input simply has an offset added to it. One terminal using this type is the TeleVideo TVI 910.

2 - coordinate is transformed into a string of ASCII digits. The DEC VT100 is a terminal with this addressing type.

3 - coordinate is transformed into a BCD byte. One terminal using this type is the ADDS-25.

When the data are displayed in the menu, integers are simply printed, and strings are displayed in a readable format, as are coordinate specifications. If a string or coordinate is not defined (I.E. - the user entered a null string for that function) a set of dashes is displayed. Additionally, a string is delimited by "" (double quotes), and coordinates are delimited by [].

APPENDIX C - ASCII CHARACTER CODES

CHAR	DEC	HEX
^@ NUL	0	00
^A SOH	1	01
^B STX	2	02
^C ETX	3	03
^D EOT	4	04
^E ENQ	5	05
^F ACK	6	06
^G BEL	7	07
^H BS	8	08
^I HT	9	09
^J LF	10	0A
^K VT	11	0B
^L FF	12	0C
^M CR	13	0D
^N SO	14	0E
^O SI	15	0F
^P DLE	16	10
^Q DC1	17	11
^R DC2	18	12
^S DC3	19	13
^T DC4	20	14
^U NAK	21	15
^V SYN	22	16
^W ETB	23	17
^X CAN	24	18
^Y EM	25	19
^Z SUB	26	1A
^[ESC	27	1B
^\ FS	28	1C
^] GS	29	1D
^^ RS	30	1E
^_ US	31	1F
space	32	20
!	33	21
"	34	22
#	35	23
$	36	24
%	37	25
&	38	26
'	39	27
(	40	28
)	41	29
*	42	2A
+	43	2B

CHAR	DEC	HEX
,	44	2C
-	45	2D
.	46	2E
/	47	2F
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B
<	60	3C
=	61	3D
>	62	3E
?	63	3F
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49
J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57

CHAR	DEC	HEX
X	88	58
Y	89	59
Z	90	5A
[	91	5B
\	92	5C
]	93	5D
^	94	5E
_	95	5F
`	96	60
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
|	124	7C
}	125	7D
~	126	7E
DEL	127	7F

APPENDIX D - SYNTAX SUMMARY

This is a concise summary of the legal syntax for the applications language. In being concise it is also very cryptic, meaning that to totally understand it will take time. It is included here as a ready reference that can be kept near the computer when making and compiling a library file of modules.

There are a number of representational conventions which are used in the following definitions:

- `|` separates choices for those parts of the syntax where more than one option is available.
- `{}` delimits syntax which may be repeated 0 or more times.
- `[]` delimits optional portions of syntax.
- `<>` delimits english explanations when syntactical definitions are insufficient. For more information on these portions of syntax, use the reference manual.
- `-` A word consisting of non-uppercase characters is an abstract identifier representing a portion of the legal syntax, usually mneumonically.
- `-` A word consisting of uppercase letters is literal text expected by the structure of the syntax at that point. In practice, this text is not case sensitive.

The general language definition follows:

```
library ==
    {module}

module ==
    module_header
    declarations
    statements

module_header ==
    MODULE modulename [formal_arguments] separator

formal_arguments ==
    (arg_list)

declarations ==
    {declaration separator}

declaration ==
    INTEGER var_list |
```

```
    LONG [INTEGER] var_list |
    DATE var_list |
    REAL var_list |
    TEXT text_list

var_list ==
    ident {,ident}

text_list ==
    textvar {,textvar}

textvar ==
    ident [OF [LENGTH] unsigned]

arg_list ==
    identifier{,identifier}

ident ==
    identifier[(dim_list)]

dim_list ==
    unsigned{,unsigned}

statements ==
    {statement separator}

statement ==
    error_command |
    file_command |
    input_command |
    output_command |
    program_control |
    miscellaneous

separator ==
    <COLON> | <CARRIAGE RETURN>


error_command ==
    RESUME [AT label] |
    RETRY |
    SET option

file_command ==
    CHD $ |
    CHECK  [f]  [k] |
    CLEAR  [f] |
    CLOSE  [f] |
    COPY  f1  [k]  TO  f2 [r] |
    COPY STRUCTURE OF f [k] TO $ |
    DELETE [f] [k] [r] |
    FIND [f] [k] [mod] |
```

```
    INSERT [f] |
    LINK f1 [k] TO f2 expression |
    LIST [f] [k] [r] |
    MARK [f] [k] [r] |
    OPEN $ [AS f] |
    REINDEX [f] [k] |
    UNLINK [f] |
    UNMARK [f] [k] [r] |
    UPDATE [f] |
    USE [f] [k]

input_command ==
    ENTER fieldname [MASK $] |
    INPUT var_list |
    SET option

output_command ==
    CLEAR FORM |
    CLEAR LINE |
    CLEAR SCREEN |
    DISPLAY fieldname [MASK $] |
    EJECT PAGE |
    LOCATE n1,n2 |
    PRINT print_list |
    SET option

miscellaneous ==
    EXECUTE $ |
    NOTE string |
    SET option |
    SHELL $ |
    [LET] let_list

program_control ==
    call |

    CASE
      statements
     {WHEN condition THEN
        statements
      ENDWHEN separator
      statements}
    ENDCASE |

    CHAIN modulename [paramlist] |

    END expression |

    END |

    EXIT |
```

```
    GOSUB label |

    RETURN |

    GOTO label |

    IF condition THEN statements [ELSE statements] ENDIF |

    LABEL label |

    LOOP statements ENDLOOP |

    QUIT |

    REDO |

    REPEAT statements UNTIL condition |

    WHILE condition DO statements ENDWHILE


general_value ==
    KEY |
    MIN (expr_list) |
    MAX (expr_list) |
    variable |
    fieldvalue |
    all

date_function ==
    TODAY |
    DATE (expression)

numeric_function ==
    ABS (n) |
    ASCII ($) |
    DUPLICATE (expression) |
    EOF (f) |
    ERROR |
    INTEGER (expression) |
    KEY PRESSED
    LENGTH ($) |
    LINE NUMBER |
    LONG (expression) |
    MARKED |
    NOT |
    PAGE NUMBER |
    REAL (expression) |
    RECORD |
    ROUND (n1,n2) |
    SIGN (n) |
    SQRT (n) |
```

```
    SUBSTR ($1,$2) |
    TRUNCATE (n1,n2) |
    VALUE ($) |

text_function ==
    CAP$ ($) |
    CHR$ ($) |
    GETKEY |
    LEFT$ ($,n) |
    LIBRARY$ ($) |
    MID$ ($,n1,n2) |
    PADCENTER$ ($,n) |
    PADRIGHT$ ($,n) |
    RIGHT$ ($,n) |
    SOUND$ ($) |
    TEXT (expression) |
    TRIM$ ($) |
    TIME

arithmetic_operator ==
    + | - | * | / | %

text_operator ==
    +

boolean_operator ==
    AND | OR | XOR

relational_operator ==
    =  | EQ |
    >  | GT |
    <  | LT |
    >= | GE |
    <= | LE |
    <> | NE

text_relational ==
    BW | CT | SL


option ==
    BOTTOM MARGIN TO n |
    DATE TO $ |
    FOOTER label_opt |
    FORM form_opt |
    HEADER label_opt |
    INPUT input_opt |
    LEFT MARGIN TO n |
    PAGE NUMBER TO n |
    PRINT print_opt |
    RIGHT MARGIN TO n |
    SCREEN toggle_opt |
```

```
    SINGLE toggle_opt |
    TOP MARGIN TO n |
    TRAP label_opt

label_opt ==
    TO label | OFF

form_opt ==
    TO $ | OFF

input_opt ==
    FROM $ | ON | OFF

print_opt ==
    TO $ | ON | OFF

toggle_opt ==
    ON | OFF

mod ==
    [APPROX][expression] |
    EXACT [expression] |
    FIRST |
    LAST |
    PREVIOUS |
    NEXT |
    RECORD n

f ==
    FILE filetag

k ==
    KEY keyname | NOKEY

r ==
    range {action}

range ==
    ALL |
    ALL FOR condition |
    ALL WHILE condition |
    PREVIOUS n [FOR condition | WHILE condition] |
    NEXT n [FOR condition | WHILE condition] |
    CURRENT

action ==
    LOWEST expression TO variable |
    HIGHEST expression TO variable |
    COUNT TO variable |
    TOTAL expression TO variable |
    PRINT print_list |
    LET let_list
```

```
print_list ==
    {[print_expr] print_sep} [print_expr]

print_expr ==
    expression | TAB(n)

print_sep ==
    <SEMICOLON> | <COMMA>

let_list ==
    assign {,assign}

assign ==
    variable = expression

fieldvalue ==
    FIELD (expression)

call ==
    CALL modulename [paramlist]

paramlist ==
    (expr_list)

expr_list ==
    expression{,expression}

condition ==
    n <THIS EXPRESSION IS INTERPRETED AS A BOOLEAN VALUE:
       n=0 is FALSE, n<>0 is TRUE>

expression ==
    n | $ | d | mask_expr

mask_expr ==
    expression MASK $ | (mask_expr)

n ==
    int_const |
    long_const |
    real_const |
    numeric_function |
    general_value |
    (n) |
    n n_binary n

n_binary ==
    arithmetic_operator |
    boolean_operator |
    relational_operator
```

```
$ ==
    text_const |
    text_function |
    general_value |
    ($) |
    $ $_binary $

$_binary ==
    text_operator |
    boolean_operator |
    relational_operator |
    text_relational

d ==
    date_const |
    date_function |
    general_value |
    (d) |
    d n_binary d

filetag ==
    identifier

modulename ==
    identifier

label ==
    identifier

variable ==
    identifier | fieldname

keyname ==
    fieldname

fieldname ==
    [filetag.]identifier

date_const ==
    text_const <THE ENCLOSED TEXT MUST BE A VALID DATE>

text_const ==
    "string" | 'string' <THE string MAY NOT CONTAIN AN
                         OCCURRENCE OF THE DELIMITER>

string ==
    {<ANY ASCII CHARACTER EXCEPT NULL>}

int_const ==
    [sign]unsigned <ITS VALUE IN THE RANGE OF AN INTEGER>
```

```
long_const ==
    [sign]unsigned <ITS VALUE IN THE RANGE OF A LONG>

real_const ==
    [sign]mantissa[exponent]

sign ==
    + | -

exponent ==
    E[sign]unsigned


mantissa ==
    unsigned[.unsigned] |
    [unsigned.]unsigned

unsigned ==
    digit{digit}

identifier ==
    alpha{alpha | digit | <UNDERSCORE> | <DOLLAR SIGN>}

alpha ==
    <ANY UPPER OR LOWER CASE LETTER>

digit ==
    <THE TEN ASCII DIGITS (0 thru 9)>
```

APPENDIX E - OPERATOR PRECEDENCE

The following is a list of operator precedence in order of least to most:

1. AND OR XOR
2. =, >, <, >=, <=, <>, BW, CT, SL, MASK
3. + -
4. * / %
5. all functions

APPENDIX F - EXTRA FEATURES

The following is a list of features and comments that are not mentioned in any other part of the documentation. None of the following features are significant; they can all be written in the standard ways used in other parts of the documentation.

Looping structure

Any of the start loop statements- **LOOP**, **WHILE**, and **REPEAT** - can be matched with any of the end loop statements - **ENDLOOP**, **ENDWHILE** and **UNTIL** - to define a loop. For example:

```
LOOP
 ...
ENDWHILE
```

is valid. (This is exactly the same as a LOOP ... ENDLOOP.) A WHILE ... UNTIL loop would then be valid and it would have two conditions, one for the WHILE and one for the UNTIL.

LET allows comma for multiple assignment

Another way of assigning a large number of values is with the comma separator. For example:

```
a=0
b=3
c$="hello world"
```

could be written as:

```
a=0,b=3,c$="hello world"
```

Multiple statement lines using colon (:)

Statements which are normally written one per line can be bunched up together using a colon to separate each statement. For example:

```
a=0
PRINT a+7
PRINT a-7
```

could be written as:

```
a=0:PRINT a+7:PRINT a-7
```

Numeric expressions can be treated as conditions

Any numeric expression can also be used as a condition in a WHILE, WHEN, IF, or UNTIL statement. If the value of the numeric expression is 0 then the condition is false. If the value of the numeric expression is not 0 then the condition is true. For example:

```
IF expression <> 0 THEN
```

can now be written

```
IF expression THEN
```

A common WHILE loop:

```
WHILE EOF(file tag) = 0 THEN
  ...
  FIND NEXT
ENDWHILE
```

can now become:

```
WHILE NOT EOF(file tag) THEN
  ...
  FIND NEXT
ENDWHILE
```

Conditionals can be treated as numbers

A conditional can be treated as a number. A conditional expression which evaluates to TRUE has a value of one while a FALSE expression has a value of zero. For example:

```
PRINT MID$("YESNO",(a$ = "N") * 3 + 1,3)
```

APPENDIX G - COMPILER, CREATE AND INTERACTIVE ERROR MESSAGES

Arithmetic operand expected
A numeric value was expected, check to see if quote marks or a missing number is the problem.

CASE missing for control structure
A WHEN or ENDWHEN was used without the CASE statement preceding it.

Close parenthesis expected
The number of open parentheses, "(", is greater than the number of close parentheses, ")".

Comma expected
A comma was expected, check for syntax errors in the statement.

DO clause expected in WHILE statement
A DO must follow the WHILE statement on the same line.

ENDCASE required to complete control structure
A CASE statement previously started the CASE structure and the matching ENDCASE is missing.

ENDIF required to complete control structure
An IF statement previously started the IF structure and the matching ENDIF is missing.

ENDLOOP required to complete control structure
A LOOP statement previously started the LOOP structure and the matching ENDLOOP is missing.

ENDWHEN required to complete control structure
A WHEN statement previously started the WHEN structure and the matching ENDWHEN is missing.

ENDWHILE required to complete control structure
A WHILE statement previously started the structure and the matching ENDWHILE is missing.

Equals symbol (=) required in assignment statement
In an assignment the equals symbol is missing.

Error in record range specification
The range specification is incorrect, for example:

```
FIND ALL NEXT
```

FIELD, HEADER or KEY must start declaration
When defining fields and keys in a definition file, the defining line must start with FIELD, HEADER or KEY.

FILE clause expected
The FILE **file tag** part of the file command is missing.

FILE must be first statement
The first statement in a definition file must be FILE **filename.**

Filename must follow FILE statement
The FILE statement must be followed by the name of the file.

IF missing for control structure
An ELSE or ENDIF was attempted before an IF statement was seen.

Illegal file name format
The file name syntax is incorrect, see IDENTIFIERS.

Illegal label format
The label syntax is incorrect, see IDENTIFIERS.

Illegal variable format. Variable must start with a letter and can contain only letters, numbers and the underline character
The variable name broke the above rules, see IDENTIFIERS.

Incorrect number of parameters for function call
Too few or too many parameters are passed to a function call, for example:

```
PRINT MID$(a$,4,3,2)
```

Label already defined
An attempt was made to state a LABEL name that was already declared somewhere else in the module.

Label not defined for jump
A GOTO, GOSUB, ROUTE ERRORS TO, RESUME AT, SET HEADER TO, SET FOOTER TO or SET TRAP TO statement is using a label that doesn't exist.

LOOP missing for control structure
An ENDLOOP was encountered when there was no incomplete LOOP structure.

Module expected
A MODULE statement must be followed by the **module name** and an optional parameter list.

Module header required before any other program statements
A MODULE **module name** statement was not found before the program statements were attempted.

No array dimensions or dimensions are not integer constants
An array variable must have at least one dimension and the dimensions must be in the form of integer constants.

Open parentheses expected
An open parentheses character -"(", was expected for the parameter list, function, etc.

Operand/operator type mismatch
The operator or the values with the operator do not match, for example:

```
PRINT LEFT$(1,2)
```

Option word expected
The option part on a SET statement is missing, or spelled incorrectly.

REPEAT missing for control structure
An UNTIL statement was encountered when there was no incomplete REPEAT structure.

Statement not legal in interactive mode
Program control (IF, LOOP, GOTO, etc.) and variable declaration statements are not allowed in interactive mode.

Syntax error, unrecognized use of command
The statement is too garbled to make sense of, check for spelling mistakes, missing parts of commands, etc.

Terminator expected; illegal use of command
The statement was trying an illegal operation, for example:

```
PRINT 3 NOT 4
```

TEXT constant expected for MASK
In a declaration file, the text following the MASK clause must be a TEXT constant.

THEN clause missing in an IF or WHEN statement
A THEN must follow the IF **condition** or WHEN **condition** statements.

To clause expected
A COPY or LINK statement has the TO part of its statement missing.

Unbalanced arithmetic expression
The number of open parenthesis do not match the number of close parenthesis, for example:

```
PRINT 3+(4*3
```

Unrecognized expression terminator
The expression finished in an incorrect manner.

Unrecognized module header format
The MODULE statement has syntax errors, or illegal characters, etc.

Unrecognized separator
A separator was not found or is incorrect, for example, a comma not found to separate items in a list of parameters.

Unterminated TEXT constant
A text string did not end with the double quote or single quote character it began with, for example:

```
PRINT "this is the
```

Variable already declared
A variable once declared in the module can not be declared again in the same module, also a variable can not be declared with the same name as a parameter.

Variable name cannot be a reserved word
The variable name can not be the same as an IMS statement word, like IF, FIND, PRINT.

WHEN missing for control structure
An ENDWHEN was encountered when there was no incomplete WHEN structure.

APPENDIX H - INTERPRETER ERROR NUMBERS

0 - **No error**
There is currently no error.

10 - **Number used where text expected**
A field or a parameter of numeric type was used in a function that wanted a TEXT type.

11 - **TEXT to DATE conversion impossible**
A TEXT value used as a date value was incorrect. For example:

```
date variable = "June 45,1234"
```

12 - **TEXT value required for the function**
A function expected a TEXT value but got something else.

13 - **A parameter not present**
A module tried to use more parameters than it was passed.

14 - **Subscript out of range**
An index value in an array was too high or too low for the size of the array.

15 - **Wrong number of subscripts**
An array was used in a statement with too many or too few subscripts or indexes. For example:

```
REAL singular(10)
PRINT singular(2,3)
```

16 - **Integer required for the operation**
An integer value is required for the operation, for example the % operator.

17 - **Overflow during numeric conversion**
Doing a INTEGER statement on a REAL value resulted in a number above the integer's range.

18 - **Date is out of range**
The value specified is too high or too low for the range of a date value.

19 - **Divide by zero error**
There was an attempt to divide by zero.

20 - **No error trap active**
A RESUME, RESUME AT, or RETRY statement was attempted when not in an error trap routine.

21 - **RETURN with no GOSUB**
A RETURN was attempted with no previous GOSUB executed.

22 - **Can't RESUME after error in END or RETURN**
An error in an END statement or a RETURN statement was trapped and in the error trap the RESUME was used to attempt to continue execution. This is invalid.

23 - **Out of memory**
The module has run out of memory. Try running the module with more memory (use the OS9 command line option to specify more memory).

24 - **Illegal operator for the operand**
There was an attempt to use an operator when the values working with it were invalid. For example:

```
PRINT REAL(10) % 4
```

would be a REAL value in a "%" operation, which is invalid.

25 - **Negative SQRT value**
A negative value was passed to the SQRT (square root) function.

26 - **Not allowed when temporary value is on stack**
An expression included a CALL statement to a module which had one of the following statements:

```
UNLINK
OPEN
CLOSE
LINK
SET FORM TO
SET FORM OFF
CLEAR SCREEN
```

for example:

```
a=b+c+CALL mod
      ...
MODULE mod
OPEN "maillist"
```

40 - **Duplicate file tag**
There was an OPEN statement using a file tag that had already been used.

41 - **Too many opened files**
An OPEN was attempted when there was already the maximum files open.

42 - **Error opening data file**
The file specified in an OPEN statement could not be opened because it was not present or not a data file.

43 - **Error opening index file**
The index file used with the data file when it is OPENed was not present or did not have permission for read and write.

44 - **Field is not present**
The field could not be found, check your spelling and ensure that the correct files are OPEN.

45 - **File tag not found**
No OPEN file has this file tag.

46 - **Incorrect key**
The current file has no key with this name.

47 - **Incorrect file operation with NOKEY**
A FIND statement with the NOKEY key option was attempted.

48 - **Valid data record required for operation**
A file command was used on an invalid record. For example:

```
DELETE RECORD 0
```

49 - **File is already linked**
A LINK was attempted on a file or a field in a file which was already linked. See LINK in the reference section.

50 - **Error opening device**
A SET command was used on a device which was not present, incorrect for the operation, etc. It could also be from an error is opening a file during a COPY statement. Check to see you are in the correct directory.

51 - **Hardware error in reading or writing**
A hardware error occurred reading or writing on the disk. This could indicate a defective disk or drive, disk not present, etc.

52 - **Module not found**
A CALL or a CHAIN to a module cannot be executed because the module can not be found, it is of the wrong type, or there is not enough memory.

53 - **End of input file**
End of file was reached during an INPUT statement. This will only occur if the standard input path has been rerouted with a previous SET INPUT FROM statement.

70 - **Internal error**

An error occurred signifying incorrect operation in IMS. Please send a hardcopy of the situation, statements, compiler listing and file data information that was used when the error occurred to **Clearbrook Software Group.**

APPENDIX I - COMMAND LINE INVOCATION

The various programs which make up IMS can be executed from the operating system prompt instead of from the main menu. The syntax to do this is as follows:

Text editor

tx **file1 file2**
or
tx **filename**
or
tx

where **file1** or **filename** is the text file to be edited and **file2** or **filename** are the default output file names.

Generate a data file

imsD **filename** [-l]

where **filename** is the file descriptor to generate one or more data files. If the -l option is selected, a listing of the compilation will be produced.

Paint a screen form

imsF **{database}**

where **{database}** is a list of data base file names (without extension) as generated from imsD.

Describe a report format

imsR **{database}**

Invocation and syntax is identical to that for imsF.

Compile an IMS module

imsC **sfilename** [-l] [-o=**ofilename**]

where **sfilename** is the name of a text file containing IMS module(s), the resulting compiled module will be a module with the same file name, unless the -o= option is selected in which case it's name will be **ofilename.** If the -l option is selected, a compilation listing is produced. The output file will be placed in the execution directory unless **ofilename** begins with a /.

Execute a compiled module

imsI **filename**

where **filename** is the name of a file containing compiled module(s).

Enter the interactive mode

imsI

Enter the CSG IMS executive

ims

APPENDIX J - FILE EXTENSIONS

IMS uses 4 character extensions at the end of files to mark what the contents are. For example, file **maillist.ida** is the data file for the maillist application. These extensions are automatic and (except for the text editor) do not need to be (and should not be) specified by the user.

The extensions are:

.ida	data file
.iin	index file
(no extension)	compiled IMS module
.isc	screen form
.ire	report form

It is recommended that for clarity an extension of **.imo** be used for a non-compiled program module and an extension of **.ide** be used for data base definition files. Because these are only recommendations, you will have to specify the extension when you invoke IMS programs which use these files.

EXAMPLES:

```
imsC menu.imo -o=menu

imsD customer.ide

OPEN "customer"

SET FORM TO "customer"
```

APPENDIX K - DATA BASE CREATOR REFERENCE

The data base creator is the program used to define and create data bases. The syntax of the specification language it expects is not a separate language, but an extension of the existing applications language. This extension is based on the data type declarations. For a description of data type declaration, refer to the section entitled **Data Types** in the reference manual.

In the following definitions, the {} delimiters indicate that the enclosed definitions may be repeated 0 or more times. The [] delimiters indicate the enclosed optional definitions. Definitions separated by "|" indicate choice of the definitions; one of the listed definitions must be used. Upper case text indicates that that text is taken literally, while lower case text is a descriptive tag for some particular definition. The general syntax of a data base descriptor file follows:

```
descriptor file        = {database_declaration}

database declaration   = file_heading
                           {data_fields}
                           {key_fields}

file_heading           = FILE identifier

data_fields            = HEADER field |
                         FIELD field

field                  = declaration [mask] [alias]

key_fields             = KEY declaration=expression [alias]

mask                   = MASK string

alias                  = ALIAS identifier
```

where

- declaration is the standard variable declaration, defined in the syntax summary (appendix D).
- expression is a standard general expression, but restricted to field variables and constant expressions. See appendix D.
- string is a text constant as defined in appendix D.

- identifier is a standard identifier. Refer to appendix D for its definition.

As indicated, the mask clause is optional. It is used to define a mask to use with the ENTER/DISPLAY commands. If this clause is missing, the mask defaults are assumed:

- for INTEGER: "######"
- for LONG: "###########"
- for DATE: "M d, Y" or the current date format
- for REAL: "**********"
- for TEXT: "***...**" (to maximum length of the field)

If the clause is present, it supercedes the default mask. This may be overridden by specifying a mask in the forms editor, and this may be superceded by specifying a mask clause in an ENTER/DISPLAY command.

The alias clause is a convenience feature. It is used to specify an alternate name by which the associated field may be referenced. Only one alias is allowed per field. Its primary use is to have a self-documenting primary field name, and also have an easily typed short alias name.

When the user invokes the database creator program on a source file that conforms to the above specifications, two files are created. These are: a data file and an index file. Their names are taken from the file heading identifier, the data file adding a **.ida** extension, and the index file adding a **.iin** extension. These two files together constitute a data base, and must be found together in the same directory in order to be used. In any programs referencing them, they are named as one unit using the file heading identifier.

Both of the files of a data base have a well defined structure. Each is composed of a dynamic number of records, the structure of each record being identical to all others in that file. This record is the base unit of information the data management routines in the interpreter use. Each record is in turn composed of a number of fields. For the data file, these fields are the data fields defined in the corresponding descriptor file, and for the index file, these are the key fields defined in the descriptor file.

The structure of a database links the two files together. They may in fact be said to be mutually dependent. For each record in the data file, there is one record for each key in the index file which is tied to it. This relation exists because the index file in fact contains the information on which the database is indexed (as the name is supposed to suggest). Thus, the data file is used

to hold the data records and maintain the structure of the database, while the index file references the data file through an internal structure organized as a b_tree. This approach to the structure of the database results in more efficient use of computer resources, particularly when the keys need to be used to find a data record.

EXAMPLE:

```
NOTE customer data file

FILE customer

NOTE keep some information about our company
HEADER TEXT company OF LENGTH 40
HEADER TEXT company_street OF LENGTH 30
HEADER TEXT company_city OF LENGTH 20
HEADER TEXT company_state OF LENGTH 3
HEADER TEXT company_zip OF LENGTH 9
HEADER TEXT company_phone OF LENGTH 10 MASK
                                        "(###)###-####"
HEADER REAL company_sales MASK "###^,###.##"

NOTE now the customer information
FIELD INTEGER customer_number MASK "##-##" ALIAS cno
FIELD TEXT name OF LENGTH 40
FIELD TEXT street OF LENGTH 30
FIELD TEXT city OF LENGTH 20
FIELD TEXT state OF LENGTH 3
FIELD TEXT zip OF LENGTH 9
FIELD TEXT phone OF LENGTH 10 MASK "(###)###-####"
FIELD DATE last_activity
FIELD REAL purchases(12) MASK "###^,###.##"
FIELD LONG credit_limit
FIELD TEXT terms OF LENGTH 10

NOTE this completes the fields
NOTE now we define the keys

KEY INTEGER number = customer_number ALIAS cno
KEY TEXT name OF LENGTH 30 = CAP$(name)
```

APPENDIX L - TEXT EDITOR REFERENCE

Tx is a general text editor that is designed to make alteration of text easy. It is a stand alone program, but is included with IMS for editing IMS source files. This text editor makes editing a great deal easier than with a word processor, and users will soon become used to its simple but powerful operation.

When **tx** is invoked, it loads the first file specified on the command line. If there is no file, the editor comes up with a blank screen with one carriage return. If a second file was specified, that file will be the default write file.

When the editor begins, the screen will display the first lines in the file. This display takes the following format: lines which are longer than the display wrap around to the next line. No word wrapping is supported, since this editor is intended as a simple program editor. The end of a text line (I.E. - a carriage return) is marked with a < character in some other intensity than the text line. If there are **n** rows on your screen, the text editor will display text on the first **n-1** rows. This allows the bottom row to be used as a status line, or to display prompts and submenus, and report errors.

The initial mode of the text editor is text insertion. Wherever the cursor resides, if a printable character is typed, it is inserted into the text at the position of the cursor, and the screen is immediately updated. In order to move the cursor, single control codes must be typed. These are outlined below. In order to perform more complex operations, such as loading a file or finding a string of text, another mode must be entered. In these other modes, the user can do no insertion of text.

The complete list of operations of the text editor follows below. (Note, the ^ symbol indicates that the control key is held down while the following character is pressed. So ^**P** means hold down the control key and press the **P** key, ^^ would be control up caret, etc.)

Abort Action ^**A,ESC**

This will stop any current action; for example, it will terminate a **FIND, BLOCK, SAVE** etc., operation and go back to the editor. NOTE: in insertion mode, ESC is used to select the I/O submenu.

Cursor keys

CURSOR UP, ^K
This will move the cursor up one line.

CURSOR DOWN, ^J
This will move the cursor down one line.

CURSOR LEFT, ^H
This will move the cursor left one character.

CURSOR RIGHT, ^L
This will move the cursor right one character.

Start HOME or ^^
If the cursor is not at the start of a line then **HOME** will put the cursor there. Then another **HOME** will move the cursor to the first line of the screen. Another **HOME** will move the cursor to the first character of the file.

End ^E
If the cursor is not at the end of a line then **^E** will put the cursor there. Then if another **^E** is used the cursor will move to the last line of the screen. Another **^E** will move the cursor to the last character of the file. In other words, three **^E**'s are usually necessary to go to the end of the file.

Next screen ^N
This will move the cursor to the next screen of text.

Previous screen ^P
This will move the cursor to the previous screen of text.

BLOCK ^B
This will display the following menu:

?,Help Cut Duplicate Paste Write

The various options are selected by typing the first letter of the word you want. These options act on a block of text which the user selects in this mode. The block is defined by using any of the previous movement commands.

CUT
This will delete a defined block from the text and put it in the block buffer, erasing what was in the block buffer before. Before this operation is used a block area must be defined. The start of the block is where the cursor was when the **^B** was pressed. The end of the block is the location of the cursor when the **CUT** option is chosen. Note that as you move the cursor the defined block is marked with low intensity or reverse

video.

DUPLICATE
This will take a defined block that has been marked and put it in the block buffer, leaving the block itself in place. Before this operation is used a block area must be defined. The start of the block is where the cursor was when the **^B** was pressed. The end of the block is the location of the cursor when the **DUPLICATE** option is chosen. Note that as you move the cursor the defined block is marked with low intensity or reverse video.

WRITE
This will give the prompt:

WRITE; file to save to:

You can then give the name of a file. If the file already exists, the computer will prompt:

Rewrite or append ? (R*/A)

Pressing the **R** key will mean that the block buffer will be written to the stated file, destroying anything else in the file. Pressing the **A** key will add to the file the block buffer contents. If the file did not exist it will create the file and write the block buffer to it.

PASTE
This will insert the contents of the block buffer, defined by a **CUT** or **DUPLICATE** operation, at the current cursor location.

?,HELP
This will display a help screen giving the important details of the editor.

Delete character DEL
The **DEL** key deletes the character on the left of the cursor.

Delete cursor character ^D
Delete the character underneath the cursor.

Undelete Character ^U
The **DEL** and **^D** operations both delete a character from the text, and store the characters in the delete buffer. Doing a **^U** causes the most recently deleted character to be put at the current cursor position.

Another ^U would cause the next character in the delete buffer to appear at the current cursor position, etc.

Line delete ^X
This will delete the entire line that the cursor is currently on, up to and including the carriage return. If the line the cursor is on is the last one in the file, the command is ignored.

Find/Replace ^F
This is for finding and replacing text. ^F will set the status line menu to:

?,Help Line Define Next Previous

The various choices are selected by pressing the first letter of each word. The choices are explained below:

DEFINE
Here you will get the prompt:

DEFINE; text to find:

as the computer waits for you to type in the text for the find or replace. Because case is not important, type in the text in any case. The computer will then prompt:

Find all occurrences ? (Y/N*)

Here a **Y** response signifies that all occurrences in the text will be found or replaced. A **N** response indicates that only one occurrence is to be found or replaced. After this, the computer prompts:

Replace text ? (Y/N*)

A **N** response indicates that a Find with no replace operation is wanted. The FIND menu will the be redisplayed. If a **Y** is entered to the prompt then a Find with replace operation is wanted. The computer prompts:

Replacement text:

and the user can then enter the text to replace the to **find** text. The FIND menu will then be redisplayed.

NEXT
This does a Find/Replace in the forward direction from the current cursor position, based on the information given in a previous FIND - DEFINE operation.

PREVIOUS
This does a Find/Replace in the backward direction from the current cursor position, based on the information given in a previous FIND - DEFINE operation.

LINE
This gives the prompt:

FIND LINE; line number:

The user then types in a line number, one or greater, of the line to find. If the line number is out of range of the file, then the prompt is redisplayed. Otherwise that line is displayed as the top line of the screen, with the cursor at the first character in the line.

?,HELP
This displays a help screen, giving the important facts about the text editor.

Input/output ESC
This displays the menu on the status line.

?,Help Indent Pass Save Load Clear Quit

To select one of the options type the first character of the word, or a question mark for HELP. The options are explained below:

INDENT
When this selection is toggled on, the text editor will automatically indent any subsequent new line typed by the user by the number of spaces on the previous line.

PASS
This gives the prompt:

PASS; Command to pass:

Here the editor is waiting for an operating system command. For example, **DIR** would be an operating system command which would do a directory. (See your operating system manual for legal commands.)

SAVE

This gives the prompt:

SAVE; save as **file** ? (Y*/N)

where **file** is the filename specified when the text editor was started. Typing **Y** will save the text in this file. Answering **N** will cause the following prompt to be displayed:

SAVE; file name:

Whatever name the file is saved under, the text editor will automatically make a backup of any file it overwrites.

LOAD

This gives the prompt:

LOAD; file to load from:

Here you can type in the name of a file, and if it exists it is loaded into the text at the current cursor position.

CLEAR

This will give the prompt:

CLEAR; are you sure? (Y/N*)

A **Y** response will delete all the text in the editor. A **N** response will cancel the CLEAR command.

QUIT

This gives the prompt:

QUIT; are you sure? (Y/N*)

If you type **Y**, the text editor finishes and all changes since the last SAVE are lost. **N** cancels the QUIT command.

Here is a synopsis of the special keys for the text editor.

Abort action	^A,ESC
Block Operation	^B
Cut	C
Duplicate	D
Write	W
Paste	P
Help	?,H
Cursor control	
Up one line	^K or cursor up
Down one line	^J or cursor down
Left one character	^H or cursor left
Right one character	^L or cursor right
Deletion	
char left of cursor	DEL
char under cursor	^D
entire line	^X
End	
End of line	^E
End of screen	^E^E
End of file	^E^E^E
Find/Replace	^F
Define	D
Forward	F
Backward	B
Line	L
Help	?,H
Input/Output	ESC
Indent	I
Save	S
Load	L
Clear	C
Quit	Q
Help	?,H
Next page	^N
Previous page	^P
Start	
Start of line	HOME or ^^
Start of screen	HOME HOME or ^^ ^^
Start of file	HOME HOME HOME or ^^ ^^ ^^
Character undelete	^U

APPENDIX M - FORMS EDITOR REFERENCE

The forms editor is a flexible, powerful method of creating and modifying single screen forms which allow formatted interactive I/O with one or more data files.

To invoke the forms editor, type **imsf** followed by a list of the data base files you need to access from the form. If you had a customer file and an invoice file and you needed information from both, type:

imsf customer invoice

Once the forms editor is invoked with the desired data base file names, no more files may be included in that form. When the program begins, a blank screen is presented to the user, with the coordinates of the cursor being displayed on the bottom line. The bottom line of the display is reserved at all times for showing information; it is not accessible to the user. At this point you can move around the screen, place text and insert and delete lines and characters. In addition, boxes may be drawn anywhere on the screen, and fields from the data records may be placed on the screen.

There are, of course, a few restrictions. Text cannot be placed over box borders or on fields. Fields and box borders cannot overlap. Aside from these restrictions, the screen can be arranged in an entirely free format.

The complete list of operations in the forms editor follows below. (Note that ^ in front of a character is viewed as the control code represented by that character.)

Abort Action ^A,ESC
This will terminate any action immediately. Since the **ESC** key is used to select the I/O actions, it is not active during text entry.

Cursor Keys

CURSOR UP ^K
Cursor moves up one line

CURSOR DOWN ^J
Cursor moves down one line

CURSOR LEFT ^H
Cursor moves left one character

CURSOR RIGHT ^L
Cursor moves right one character

Next Line **^M**

The cursor will be moved to the start of line below the line the cursor is currently on.

Start **HOME or ^^**

If the cursor is not at the start of a line, HOME will put the cursor there. Otherwise the cursor will be placed at the top left corner of the screen.

End **^E**

If the cursor is not at the last column in the line, **^E** will place it there. Otherwise the cursor will be placed at the bottom right corner of the screen.

Delete Character **DEL**

Pressing the **DEL** key will delete the character on the left of the cursor, shifting the remainder of the text and fields to the right of it left one character. If a box border intersects that line, the text and fields are shifted only up to that box.

Delete Cursor Character **^D**

This deletes the character the cursor is over. The text and fields are again shifted as explained for DEL above.

Delete Line **^X**

This removes the line the cursor is currently on, moving the text, fields and boxes below that line up one line. It will not allow fields or boxes to be deleted from the screen.

Insert Character **^C**

This shifts the text from the cursor to the first box border or end of the line to the right by one character. A space is inserted at the cursor. **^C** is the complement action to **^D.**

Insert Line **^I**

This shifts all text, fields and boxes below the current line down by one line. A blank line is then inserted in the current line. It will not allow fields or boxes to be shifted off the screen. **^I** is the complement action to **^X.**

Field Access ^F

The following menu will be displayed when **^F** is pressed:

?,Help Add Delete Mask Info

The options are then selected by pressing the key of the first letter of each word.

Add

Selecting the add option allows the user to add a data record field to the screen form. When a field is selected, it will be displayed on the screen as the mask for that field in half-intensity or reverse video.

When the add option takes control, it will present the user with a list of all fields available. They will be presented as the data base creator declaration for that field, one field per line. If there are more fields than display lines, only the first field declarations will be displayed. To select a field, the up and down cursor keys may be used to move the cursor over top of the desired field; if there are more fields than lines, the display will be scrolled to allow access to all fields. When the cursor is on the desired field, **RETURN** or **ENTER** must be pressed. This will select the field and return to text entry.

Delete

For this option to work, the cursor must be on a field. If it is, the computer will prompt:

DELETE; are you sure ? (Y/N*)

Answering **Y** will **deselect** it and remove the mask from the screen.

Mask

For this option, the cursor must be on a field. If it is, it will allow the user to specify a different mask to be used. The computer will prompt:

MASK; new string:

The mask is not accepted if it overlaps borders or fields.

Info

This option displays the data base creator declaration of the selected field which the cursor is on. It also displays the coordinates of where on the screen the field starts.

?,Help
This will display help screen giving important details of the forms editor.

Box Drawing ^B
This will display the following menu:

?,Help Draw Erase

The various options are then selected by pressing the first character of each word. The options are:

Draw
With this option, the user can draw a box almost anywhere on the screen. The upper left corner of the box will be where the cursor was on the screen when this option was selected. To define the limits of the box, the user then moves the cursor around using any of the cursor movement keys. The bottom right corner of the box is defined as being the current position of the cursor. To draw the box with those limits, **RETURN** or **ENTER** must be pressed.

Erase
This option allows any box on screen to be removed. In order to work, the cursor must be anywhere in the border of a box. The computer will then prompt:

ERASE; are you sure ? (Y/N*)

Answering **Y** will erase the box, and an answer of **N** will cause the request to be ignored.

?,Help
This will display help screen giving important details of the forms editor.

Input/Output ESC
Selecting this option will display the menu:

?,Help Pass Clear Quit Save Load Output Generate

To select an option, press the letter the word begins with. The various options are:

Pass
This allows a command to be passed to the operating system for processing. The computer will prompt:

PASS; command to pass:

Clear

This option will delete all the text on the screen, erase all boxes, and deselect all fields. Before doing so, the computer will prompt:

CLEAR; are you sure ? (Y/N*)

Answering **N** will cause the clear option to be ignored.

Quit

The computer will prompt:

QUIT; are you sure ? (Y/N*)

Answering **Y** will terminate the current session of the forms editor. Answering **N** will cancel this option.

Save

This will give the prompt:

SAVE; file name:

The desired name of the screen form should be typed in. If the file already exists, the computer will prompt:

Rewrite ? (Y/N*)

Answering **Y** will replace that file with this screen form. A **N** response will cancel the save.

Load

This will give the prompt:

LOAD; screen form file:

Type the name of the screen form desired. The forms editor will then load it in, forgetting all previous text typed, boxes drawn and fields selected. Note that this new form must have been made with the same data files as were selected for the current forms editor session.

Output

This will display the prompt:

OUTPUT; destination device:

This option will write an image of the screen and a list of the selected fields to the specified destination.

Generate

Selecting this option will cause the forms editor to generate a simple program in the applications language which will use any screen form desired and data files specified when the current session of the forms editor was invoked. It will allow simple entry, editing and maintenance of the data files. For a complete description of this generated program, refer to lesson one in the tutorial. When this option is selected, the computer will prompt:

GENERATE; name of output file:

If there is more than one data file, the program will allow entry of data into only one of the data files; thus the computer will prompt:

Primary data file:

The primary data file is the one you wish to maintain. The program generator will generate ENTER statements for fields in this file and DISPLAY statements for fields in other files.

Finally, the computer will prompt:

Form to use:

You should enter the name that you gave the form when you SAVEd it.

?,Help

This will display the help screen giving important details of the forms editor.

APPENDIX N - REPORTS EDITOR REFERENCE

The reports editor is a program which allows a user to define a report in a highly visual and interactive fashion. The user "paints" the format of the desired report on the screen in a manner very similar to the forms editor. This format, known as a report form, may be saved, then loaded at a later session and edited.

In order to perform the report as defined by a report form, the user must generate a program to do the task. This program is generated by the reports editor in the CSG IMS applications language (you will need to compile it), which allows the user to customize a report. The report generated by the reports editor is sufficiently powerful to accomplish most reasonable tasks without the need to modify the generated report.

The report editor is invoked in a way similar to the form editor. The command name is followed by a list of data files which the report will access.

imsR customer invoice detail

This will invoke the report editor. The report can print any information from the files **customer**, **invoice** or **detail.**

The reports editor begins with a blank report form. If the user has saved a previous report form, it may be loaded at this point. A report form as represented on the screen, has a clearly defined structure. It consists of several sections; each section has a heading which is highlighted in some manner. Immediately following this heading, which occupies an entire line, the user may insert and delete lines, for purposes of displaying text and fields.

These sections are:

HEADER - page header definition (printed at the top of each page).
FOOTER - page footer definition (printed at the bottom of each page).
PRIMARY FILE - data listing format definition for the primary file.
PRIMARY SUBTOTALS - definition of subtotals format for the primary file.
TOTALS - definition of totals reporting format. Printed at the end of the report.
END OF REPORT - end of report form marker.

When there are multiple data bases, and the user has linked two or more of them together, for each linkage the PRIMARY FILE and PRIMARY SUBTOTALS sections are duplicated and appropriately identified.

Once the reports editor has started, the user may move the cursor about using the commands described below. Text and fields may be entered in any non-heading lines desired (a non-heading line is inserted with the ^I key). Each section represents a portion of the report as it will be printed out by the generated program. The report form (except for the highlighted headings) in fact represents a partial page as it will appear on the printer. For that purpose, there is a line length of 192 characters. Text which appears in each section is, at runtime, printed at the column where it starts in the report form. Likewise with fields.

When the generated report is processing a record from the primary file, it prints it according to the format given by the PRIMARY FILE section. All records with the same key are printed before subtotals are printed as defined in the PRIMARY SUBTOTALS section. This entire process is repeated for all records in the data base.

If there are secondary files which have been linked, they are processed in an almost identical fashion to that just described for the primary file. The primary difference is that secondary files are implicitly indexed through a LINK statement. If there is a file secondary to this one, the process is again applied, in a recursive fashion. By this method, it is possible to have an arbitrary depth of file-linkage hierarchy.

The keystroke commands the reports editor accepts are described below. Note that a caret placed in front of a character indicates the corresponding control character. I.E. - ^M is CARRIAGE RETURN or M typed when the CTRL key is being pressed down.

Abort Action ^A,ESC
This will terminate any action immediately. Since the **ESC** key is used to select the I/O actions, it is not active during text entry.

Cursor Keys
CURSOR UP ^K
Cursor moves up one line
CURSOR DOWN ^J
Cursor moves down one line
CURSOR LEFT ^H
Cursor moves left one character

CURSOR RIGHT ^L
Cursor moves right one character

Next Line ^M
The cursor will be moved to the start of line below the line the cursor is currently on.

Start HOME or ^^
If the cursor is not at the first column of the screen, HOME will will place it there. Otherwise, if the cursor is not at the top left corner of the screen, it will be placed there. Otherwise, if the cursor is not in the first column of the current line, it will be placed there. Finally, if none of the above cases is true, the cursor will be placed in the first column of the first line of the report form.

End ^E
If the cursor is not at the last column on the screen, **^E** will place it there. Otherwise, if the cursor is not at the bottom right corner of the screen, it will be placed there. Otherwise, if the cursor is not on the last column (column 192) of the current line, it will be placed there. Finally, if none of the above cases is true, the cursor will be placed in the last column of the last line of the report form.

Delete Character DEL
Pressing the **DEL** key will delete the character on the left of the cursor, shifting the remainder of the text and fields to the right of it left one character.

Delete Cursor Character ^D
This deletes the character the cursor is on. The text and fields are again shifted as explained for DEL above.

Delete Line ^X
This removes the line the cursor is currently on, moving the text and fields below that line up one line. It will not allow fields to be deleted from the form.

Insert Character ^C
This shifts the text from the cursor to the end of the line to the right by one character. A space is inserted at the cursor. This is the complementary action to **^D**.

Insert Line ^I

This shifts all text and fields below the current line down by one line. A blank line is then inserted in the current line. This is the complementary action to **^X**.

Change Borders ^B

This displays the following menu:

?,Help Left Bottom

Left

This displays the prompt:

Old value for left margin=**n**. Enter new value:

At this point the user may enter a new value for which column on the printer the left margin is to begin. Typing the RETURN or ENTER key leaves the value unchanged.

Bottom

This displays the prompt:

Old value for bottom margin=**n**. Enter new value:

At this point the user may enter a new value for which row on the printer the bottom margin is to begin. It defines where the footer begins on the page. Typing the RETURN or ENTER key leaves the value unchanged.

?,Help

This will display the help screen giving important details of the forms editor.

I/O Commands ESC

This displays the following menu:

?,Help Pass Save Load Clear Quit Generate

To select an option, press the letter the word begins with. The various options are:

Pass

This allows a command to be passed to the operating system for processing. The computer will prompt:

PASS; command to pass:

Clear

This option will delete all the text on the screen, erase all boxes, and deselect all fields. Before doing so, the computer will prompt:

CLEAR; are you sure ? (Y/N*)

Answering **N** will cancel the clear option.

Quit

The computer will prompt:

QUIT; are you sure ? (Y/N*)

Answering **Y** will terminate the current session of the forms editor. Answering **N** will cancel this option.

Save

This will give the prompt:

SAVE; file name:

The desired name of the report form should be typed in. If the file already exists, the computer will prompt:

Rewrite ? (Y/N*)

Answering **Y** will replace that file with this screen form. A **N** response will cancel the save.

Load

This will give the prompt:

LOAD; report form file:

Type the name of the report form desired. The reports editor will then load it in, forgetting all previous text typed and fields selected. Note that this new form must have been made with the same data files as were selected for the current reports editor session.

Generate

Selecting this option will cause the reports editor to generate a program in the applications language which will, when compiled and invoked, print out a report based on the information given in the currently loaded report form. This program will contain all the necessary code to print a report after the module is compiled. The program is generated such that it is modular and easy to modify if the user so wishes. For a complete description of this generated program, refer to lesson one in the tutorial.

When the GENERATE option is selected, the computer will prompt:

GENERATE; name of output file:

The computer will then ask:

Index the primary data base ? (Y*/N)

Answering **N** will cause the primary data base to be traversed in sequential record number order. Answering **Y** will cause a number of further requests by the computer.

The first is that the user must select a key by which the data base will be indexed. This is done with a selection list, which is described more completely in the Function Menu section. Secondly, when the key is selected, the user must type an expression which that key must match exactly.

The generation of the program will then procede. Remember to compiler the program before trying to execute it.

?,Help

This will display the help screen giving important details of the forms editor.

Function Menu ^F

This is a somewhat more complex menu command. Exactly what menu is displayed depends on what sort of line the cursor is in. Additionally, there are several levels of menus. Once a primary menu has come up, the desired option is selected by typing the first character of that word. Secondary menus are treated identically to primary ones; they are displayed and used in the same

manner. In the following descriptions, the primary menus are outlined, then secondary menus, and finally the various options in those menus are explained. Additionally, selection lists are defined at the end of this section.

Primary Menus

If the cursor is in a HEADER, FOOTER, PRIMARY SUBTOTALS or END OF REPORT heading line, ^F has no effect; it is ignored.

If the cursor is in a PRIMARY FILE heading line, the following menu is displayed:

?,Help Link Remove

If the cursor is in the TOTALS heading, the following menu is displayed:

?,Help Link

If the cursor is in the text portion of the HEADER or FOOTER sections, the following menu is displayed:

?,Help Number Today Print Sum Delete Mask Info

If the cursor is in the text portion of a PRIMARY FILE section, the primary menu is:

?,Help Print Delete Mask Info

If the cursor is in the text portion of a SUBTOTALS or the TOTALS section, the primary menu is:

?,Help Print Sum Delete Mask Info

Secondary Menus

Selecting **Link** allows the user to define secondary files to which the primary file is linked. The reports form editor in fact supports any arbitrary complexity and depth of file linkage, not just to a secondary file. This is currently the only secondary menu option. Its menu is:

?,Help Prime Second Key Expr Done

Options

Number

This option will place a mask of "###" on the screen where the cursor is. When the generated report is executed, the current page number will be printed out. It is treated identically to a data field.

Today

This option is similar to the **Page** option, with the difference that the current date is printed, rather than the current page number.

Print

This option allows the user to place a field where the cursor resides in the report form. This is done with a selection list (explained more completely below), where each item in the list is a data field displayed in the format in which it was declared in the data base descriptor file. In the generated report, this field will be printed out at the appropriate time in the column it was placed on in the report form.

Sum

This option operates identically to the **Print** option in the way in which a field is selected. It allows the user to define a field which will be summated. The result of the summation will be printed out as described in the **Print** option. If the cursor is in a SUBTOTALS section, after the result of a summation is printed, it will be reinitialized to 0. NOTE: summing a TEXT or DATE field will do a count of the number of records.

Delete

For this option to work, the cursor must be on a field. If it is, the computer will prompt:

DELETE; are you sure ? (Y/N*)

Answering **Y** will remove the mask from the form. Answering **N** will cancel the option.

Mask

For this option, the cursor must be on a field. If it is, it will allow the user to specify a different mask to be used. The computer will prompt:

MASK; new string:

The mask is not accepted if it overlaps any fields.

Info

This option displays the data base creator declaration of the field which the cursor is on.

Prime

This option allows the user to select which file of those loaded will be the primary data base in a LINK statement. It does this with a selection list, where each item in it has the form:

FILE **filename**

where **filename** is the name of the data base. If a primary file has already been selected, it is identified.

Second

This option operates identically to the **Prime** option, except that it allows the user to select which file will be secondary to the primary file.

Key

Selecting this option allows the user to specify by what key the secondary file will be indexed. This is done with a selection list, in the same manner as the **Print** option. If a key has already been selected, it is identified.

Expr

This option will prompt the user for an expression made of data fields of the primary file with the following query:

Search expression:

This expression will be used in a LINK statement, in order to exactly match the key in the secondary file. If an expression has already been entered, the computer will first prompt:

Replace "**expr**" (Y*/N) ?

where **expr** is the previous expression.

Done

This option will install the information given in previous **Prime**, **Second**, **Key**, and **Expr** options. This installation takes the form of two

new sections being created; one displays the linkage information, the other displays the title SUBTOTALS. These are a replication of the PRIMARY FILE and PRIMARY SUBTOTALS sections in function. If the information is incomplete or not permissible because of circular file linkages, this option will be cancelled.

Remove

This option will remove the file linkage that the cursor is over. If the cursor is on a valid linkage definition the computer will prompt:

REMOVE; are you sure ? (Y/N*)

answering **Y** will remove it, answering **N** will cancel the request to remove the file linkage.

?,Help

This will display help screen giving important details of the forms editor.

Selection Lists

A selection list is a method of querying the user for a specific item from a sometimes random or dynamic list of items. The items in the list are displayed in a column, one item per row. If there are more items than rows, the user may scroll through the list to the unlisted portions of the selection list.

Items in the list may be marked. If an item is not in normal video, it is not selectable. Otherwise, it may be selected.

To select an item, the cursor must be placed on the desired item, at which point the **RETURN** or **ENTER** key must be pressed. To place the cursor on that item, it is moved down the list with the CURSOR DOWN key and up the list with the CURSOR UP key.

APPENDIX O - IMPORTING DATA

Data imported by IMS must be in the form of ASCII text. To import the data, a short IMS module will have to be written. Examples of two different methods are:

1. Each text line contains one piece of data. This is a common method used for mailmerge files for word processors and would look like:

```
John Doe
123 Main Street

Anywhere

USA
00000
Clearbrook Software Group
446 Harrison Street
PO Box 8000-499
Sumas
WA
USA
98295
  .
  .
  .
```

Here each "record" consists of 7 lines (records) of text. A general purpose program to import this type of data would be:

```
NOTE A program to import data from a text file
NOTE in which each line is a data field.  Up to
NOTE 40 fields per record can be read.

MODULE importlines

TEXT fields(40),none,file
INTEGER i,lines

PRINT "Enter the name of the IMS file to import
      to: ";
INPUT file
OPEN file

PRINT "Enter the name of the text file to import
      from: ";
INPUT file

REPEAT
   PRINT "How many lines in each text file
```

```
            record? ";
   INPUT lines
UNTIL lines>0 AND lines<=40

LIST STRUCTURE
i=0;
WHILE i<lines DO
   i=i+1
   PRINT "To which field (name) does line ";i;
            "correspond to (type NONE if none)? ";
   INPUT fields(i)
ENDWHILE

SET INPUT FROM file
SET INPUT ON

LOOP
   i=0
   WHILE i<lines DO
      i=i+1
      IF CAP$(TRIM$(fields(i)))="NONE" THEN
         INPUT none
      ELSE
         INPUT field(fields(i))
      ENDIF
   ENDWHILE
   INSERT
ENDLOOP

END

LABEL trap
NOTE this is the error handler

IF ERROR=53 THEN NOTE end of input file
   PRINT "Finished"
ELSE
   HELP ERROR
   RESUME
ENDIF

END
```

2. Each line of the text file is a record, the fields of each record start at a fixed offset from the beginning of the record. Fields are a fixed size.

```
John Doe            123 Main Street   Anywhere  USA   00000
Clearbrook Software 446 Harrison St.  Sumas WA  USA   98295
  .
  .
  .
```

Here each "record" consists of 1 line of text. A general purpose program to import this type of data would be:

```
NOTE A program to import data from a text file
NOTE in which each line is a record.  Up to
NOTE 40 fields per record can be read.  The
NOTE text line may be no longer than 255
NOTE characters.

MODULE importline

TEXT fields(40),none,file
TEXT line OF LENGTH 256
INTEGER offset(40),fsize(40)
INTEGER i,flds

PRINT "Enter the name of the IMS file to import
        to: ";
INPUT file
OPEN file

PRINT "Enter the name of the text file to import
        from: ";
INPUT file

LIST STRUCTURE
flds=0;
WHILE flds<40 DO
   flds=flds+1
   PRINT "What is the name of IMS field number ";
flds;", press ENTER when done: ";
   INPUT fields(flds)
   IF fields(flds)="" THEN
      flds=flds-1
      EXIT
   ENDIF
   PRINT "In which column does ";fields(flds);
           " start (1 is first)? ";
   INPUT offset(flds)
   PRINT "What is the size of the field? ";
   INPUT fsize(flds)
```

```
ENDWHILE

SET INPUT FROM file
SET INPUT ON

LOOP
   i=0
   INPUT line
   WHILE i<flds DO
      i=i+1
      field(fields(i))=MID$(line,offset(i),
                                              fsize(i))
   ENDWHILE
   INSERT
ENDLOOP

END


LABEL trap
NOTE this is the error handler

IF ERROR=53 THEN NOTE end of input file
   PRINT "Finished"
ELSE
   HELP ERROR
   RESUME
ENDIF

END
```

APPENDIX P - Exporting Data

Exporting data to ASCII text files is an easier proposition than importing data. Quite simply, the reports form editor may be used to create both type of import files described in the previous section.

1. To create text files having one piece of (left justified) data per line, invoke the reports form editor with the data base file name from which you wish to export data. Next, create a report form which has no headers, footers, totals, or subtotals, and the left and bottom margins having a value of 0. In the PRIMARY FILE section, insert as many lines as fields you wish to export, then place one field, left justified, on each line. After this is complete, you should save the report form. Then simply generate a report program. This program, when compiled, will export the data to any file you specify at runtime.

2. To create text files where the data is stored one record per text line, and the data fields are column aligned fields in the text line, repeat the procedure describe above, with the following difference. In the PRIMARY FILE section insert only one line and place the fields on this line as opposed to aligning them vertically. This program, when compiled, will export the data to any file specified at runtime.

CLEARBROOK SOFTWARE GROUP INC.
SOFTWARE LICENSE AGREEMENT

LICENSE: Clearbrook Software Group Inc. owns the enclosed software program and all copyrights and other rights to it. Clearbrook Software Group Inc. grants you a non-exclusive license to use the enclosed software program subject to the terms and restrictions below.

RESTRICTION OF USE: You may use the program on a single computer. In a computer network, a separate license is required for each computer in the network on which the software is to exist. You may modify the program or merge it into another program but all such portions remain subject to the restrictions of this license.

RESTRICTION ON COPYING: You may copy the program for back-up or archive purposes provided you include the copyright notice and serial number on the copies. You may not copy any part of the documentation for this program.

RESTRICTION ON TRANSFER: You may physically transfer the program from one computer to another provided that the program is used on only one computer at a time. You may transfer this license as long as the person to whom you transfer receives all copies of the program and documentation and agrees to be bound by the terms of this license and notifies Clearbrook Software Group Inc. in writing.

TERM: This license continues in effect until terminated. You may terminate the license by destroying all copies of the program and documentation and notifying Clearbrook Software Group Inc. in writing.

LIMITED WARRANTY: If you send the Registration Card to Clearbrook Software Group, free updates will be provided to you for a period of one year from the date of purchase. It is your responsibility to notify Clearbrook Software Group Inc. of any defect you find in the program so that we may correct the defect and send you an update. After a period of one year from the purchase date, updates must be purchased for $10US plus shipping charges.

LIMITATION OF LIABILITY: Clearbrook Software Group Inc. will not be liable for any direct, incidental or consequential damages resulting from the use of the program.

Your use of the program or completing and returning the enclosed registration card acknowledges that you have read this license and agree to be bound by its terms.

Retain this copy for your records.

CLEARBROOK SOFTWARE GROUP INC.
ABBOTSFORD, BRITISH COLUMBIA