TRS-80®

UNIFY®
REFERENCE MANUAL

Radio Shack®

10 9 8 7 6 5 4 3 2 1

CONTENTS

1. INTRODUCTION 11
The UNIFY Environment 13
Installation 14
Files 25
Directories 27
Multiple Databases 29
Parameters and Environment Variables 3Ø
Termcap 33
Required Terminal Functions 33
Optional Terminal Functions 34
Keyboard Mapping 35
Setup 36

2. MENUH -- MENU HANDLER 37
Menus 38
Users 4Ø
Help Documentation 42
Executables 43
Programs 45
Program Execution via the Menu Handler 46
Using Unify as the Shell 5Ø
MENU Maintenance Screens 52
Executable Maintenance 52
Menu Maintenance 57
Group Maintenance 6Ø
Employee Maintenance 65
Enter Help Documentation 71
Program Loading 72
System Parameter Maintenance 78
Menu Handler Reports 81
Executable Listing 81
Menu Listing 82
Group Listing 84
Employee Listing 84
Print Help Documentation 85

3. UNITRIEVE -- DATABASE MAINTENANCE 87
Schema Maintenance 88
Record Maintenance 89
Field Maintenance 92
Schema Listing 97
Database Maintenance 1Ø3
Create Database 1Ø3
Reconfigure Database 1Ø6
Secondary Index Maintenance 111
Volume Maintenance 113
Security 119
Field-level Security 119
Process Field Passwords 122
A Secure Unify Environment 123
Statistics 124
Database Statistics 124
Hash Table Statistics 126
Operational Utilities 128
Write Database Backup 128
Read Database Backup 131
Usave 133
Hash Table Maintenance 136
Explicit Relationship Maintenance 136
Database Load 137

4. SFORM -- SCREEN FORM DEVELOPMENT TOOL 143
SFORM Maintenance 145
Test Screen 15Ø
Process Screen 151
Screen Reports 153
Restore Screen 155
Screen List 157
Create Default Screen Form 158

5. ENTER -- DATA ENTRY/QUERY BY FORMS 163
Registering an ENTER Screen 165
Using ENTER Screens 168
ENTER Data Entry 168
ENTER Query by Forms 169

Customizing ENTER 174
Theory of Operation 174
Custom ENTER Examples 184
Custom ENTER Functions 188
Loading ENTER 197
Multiple ENTER Executables 2ØØ

6. SQL -- QUERY LANGUAGE 2Ø5
SQL Query Facilities 2Ø6
Help Features 2Ø7
Select From Clause 21Ø
Where Clause 212
Logical Expressions and Operators 213
Not Conditions 217
Set Inclusion 218
Unique Operator 22Ø
Arithmetic Expressions 222
Order By Clause 224
Aggregate Functions 226
Group By Clause 228
Nested Queries 231
Having Clause 234
Join Queries 238
General Join 238
Self-Join 243
Variable Queries 245
UNIFY SQL Extensions 248
Editing Query Statements 248
Executing Stored Queries 249
Executing TRS-XENIX Commands 25Ø
Saving Data to TRS-XENIX Files 25Ø
Loading Data from TRS-XENIX Files 251
Reference 254
Keywords 254
Error Messages 255
Formal Grammar Syntax 269
Query Statements 27Ø
Commands 282
Help Commands 285

7. LST -- LISTING PROCESSOR 287
Selecting Records 289
Running the Selection Processor 289
Selection Syntax 289
Expressions 29Ø
Select 292
Remove 293
Call 294
Copy 295
List 296
Unlock 297
Report 298
Listing Records 299
Running the Listing Processor 299
Listing Processor Syntax 3ØØ
Expressions 3ØØ
List 3Ø2
Sort 3Ø6
Total 3Ø7
Go 3Ø8
Print 3Ø9
Nohead 31Ø
Unlock 311

8. DATABASE TEST DRIVER 313

9. C LANGUAGE INTERFACE 317
Compiling and Loading Programs 32Ø
UNIFY Error Handling 324
Record Functions 338
access 339
accsfld 34Ø
addrec 341
closedb 343
delete 344
endtrans 345
fldesc 346
gfield 348
loc 349

lockrec 35Ø
opendb 352
pfield 353
seqacc 355
setloc 356
startrans 357
ulockrec 358
Selection Processor Functions 36Ø
closesf 361
clrfitm 362
clrsitm 363
entsitm 364
frstsel 366
mtchitm 367
nextsel 368
opensf 369
prevsel 371
selsort 372
sfncitm 374
ufsel 375
unisel 376
uqmtch 378
uqsrch 38Ø
Explicit Relationship Functions 382
clrset 383
faccess 384
makeset 385
nextrec 387
prevrec 388
samerec 389
setsize 39Ø
sfrstrec 391
slastrec 392
snextrec 393
sprevrec 394
unisort 395
Secondary Index Functions 4Ø1
btnext 4Ø2
btsrch 4Ø3
closbt 4Ø4
fldidxd 4Ø5
opnbts 4Ø6

Buffered Sequential Access Functions 407
bgfield 408
bseqacc 409
bsetloc 410
bfaccess 411
iniubuf 412
Terminal I/O Functions 413
cleancrt 414
clearscr 415
dsply 416
erasprmp 417
gdata 418
gtube 420
inbuf 422
input 423
loadscr 424
outbuf 425
output 426
pdata 427
prmp 428
prtmsg 429
ptube 430
sfldesc 431
yorn 433
Printer I/O Functions 434
flush 435
oblank 436
obuf 437
odata 438
pform 439
prstr 444
Utility Functions 445
cfill 446
clr_crt 447
eras_ln 448
glob 449
ivcmp 450
kdate 451
kday 452

keybrd 453
lastchr 454
len 455
lock 456
mv_cur 459
oprf 46Ø
priamd 461
ptct_crt 463
ptct_wrt 464
qmove 465
scomp 466
setcook 467
setraw 468
strcmp 469
unlock 47Ø
valchar 471
valstr 472

APPENDIX A -- R/M COBOL INTERFACE 473

APPENDIX B -- LIST OF VI EDITOR COMMANDS 5Ø1

APPENDIX C -- ERROR MESSAGES 511

APPENDIX D -- MENU MAP 537

INDEX 543

INTRODUCTION

The Unify Reference Manual provides specific reference information on using Unify. The index at the end of the manual is useful for locating a specific subject. While some sections give examples of usage, the Unify Tutorial Manual provides more detailed examples of using Unify with an applications development project. It is the best place to start if you have never used Unify.

This manual is divided into nine sections, each of which discusses a particular aspect of Unify. Section 1 introduces the manual and gives you an overview of the Unify installation process and environment -- the directories, files, and environment variables you need to set up before using Unify. Section 2 describes your primary interface to Unify, MENUH. Section 3 explains the database maintenance utilities that create and maintain the schema and the database itself.

Section 4 discusses SFORM, the screen form tool you use to create and maintain data entry screens. Section 5 describes ENTER, a general purpose data entry program that can drive SFORM screens. You may also use ENTER as a query by forms interface to the database or as the basis for your own custom data entry screens.

Once data has been entered, Unify's SQL (pronounced see-quel), described in Section 6, extracts and presents it in a meaningful format. Unify also has a simpler query interface program, the Listing Processor, which is described in Section 7. The Listing Processor is a selection and formatting language that lets you produce simple file listings with totals, subtotals, and column headings. The Database Test Driver, a simple database editor designed primarily for programmers, is described in Section 8.

Section 9 presents the procedural host language interface tools that let you write your own custom applications. The host language interface described in Section 9 is designed for C language programs. Appendix A describes the interface to Ryan-McFarland COBOL. Any TRS-XENIX language that can make calls

to C functions (such as Pascal and Fortran) can also be used with the Unify host language interface.

The Unify Environment

Unify is a collection of more than 2Ø different integrated programs that let you create and modify applications systems for storing and retrieving data. Integration is primarily through the Unify menu handler, MENUH, which lets you select the particular program you want to run at any time.

The menu handler is a friendly alternative to the TRS-XENIX shell and is oriented toward letting unsophisticated computer users develop and operate their own applications. Each Unify tool exists only to simplify the jobs of applications development, maintenance, and operation.

The tools you use most often in developing a project are Schema Maintenance, Create Database, Database Load, Screen Form Maintenance, and SQL. If you are developing custom software, you will probably make extensive use of the Host Language Interface. Once your system is up and running, you will most often use ENTER, in both data entry and query by forms mode, and SQL.

The remainder of this chapter is a collection of miscellaneous items not discussed elsewhere in this manual but useful for conveniently operating Unify applications. The explanations that follow describe the Unify installation process, Unify files, general directory layout, shell environment variables, and the use of termcap, the terminal capabilities database.

Although operating Unify requires little or no knowledge of TRS-XENIX, the shell, or programming, the information in the remainder of this section is more demanding. You should be familiar with directories and the shell, and for the discussion of termcap, with terminal operation from a programming point of view.

Since this information deals in some way with the entire system, references to other sections of the manual are numerous. If you are not yet familiar with Unify, you might not fully understand the topics discussed. It is nevertheless a good idea to read this section for orientation before using the system.

When you are more familiar with Unify, review this section to clarify the ways in which you can apply the information.

INSTALLATION

The following conventions are basic to Unify:

<ENTER> Terminate all lines typed in on the keyboard by pressing <ENTER> unless other instructions are given. <ENTER> moves the cursor forward.

<F1> Moves the cursor backward.

<F2> Stores a record.

Note: On the Model II/12/16, the function keys (<F1>, <F2>) are located either above or beside the numeric key pad. On the DT-1, hold down <SHIFT> and press <7> to initiate the <F1> function and hold down <SHIFT> and press <9> to initiate the <F2> function.

_ A box or an underscore on the screen that represents the cursor.

To install Unify, follow these steps:

1. Turn on all peripherals and then turn on the computer.

2. When your screen displays the **login** prompt, type:

   root <ENTER>

3. At the root prompt (#), type:

   install <ENTER>

4. The screen shows the Installation Menu, which contains these prompts:

1. to install
q. to quit

Type 1 **<ENTER>** to install Unify.

5. The following prompt appears:

Insert diskette in Drive Ø and press <ENTER>

Insert your TRS-XENIX Unify Diskette 1 in Drive Ø and press <ENTER>. The screen then displays the name of the package, the version number, and the catalog number.

Unify is executed from a script named unify that is located in the directory /usr/bin. If /usr/bin/unify already exists and Unify has never been installed on the system, the installation is terminated at this point with the following error message:

> There is a file named /usr/bin/unify, although UNIFY has never been installed on this system. Please remove or rename this file and try the installation again.

Another possible error message at this point occurs if /usr/bin/unify does not exist, yet the logbook indicates Unify has been installed previously:

> The file /usr/bin/unify was not found, although /etc/logbook indicates a version of UNIFY is currently installed. Please recover your disk files and try the installation again.

To alleviate this problem, you may either attempt to recover /usr/bin/unify or delete the entry in /etc/logbook.

If Unify has been previously installed on the system and you attempt to install an older version or the same version, the following message is displayed:

> Warning. Continuation of this process will replace the current UNIFY version with an older version.

If you choose to continue the installation process, an additional warning message appears:

> The following current UNIFY directories will be removed or replaced upon successful completion of this process if present.

The screen then lists the directories containing the programs, libraries, and tutorial if present.

Be sure you have made a backup of the current database and copied all the screens, user help documentation, and application programs you wish to save from the current version, before continuing the installation by typing y. Pressing any other key causes the installation process to terminate without any destruction to the existing database.

6. If Unify has not been previously installed or if you elect to continue the installation of a new version, the following screen appears:

UNIFY COMPONENTS TO BE INSTALLED

#	Name	Free Space Required Runtime	Development	Disk #
1.	Programs (required)	xxxx	xxxx	
2.	Libraries (required)	xxxx	xxxx	
3.	Tutorial (optional)	xxxx	xxxx	

DISK CONFIGURATIONS

#	Name	Free Space Available (blocks) Now	After Installation
1.	/	xxxxx	

The various components of Unify are displayed at the top of the screen. (An indication is given to indicate whether the component is required or optional.) You can either install Unify as a Runtime system, or if your system includes the TRS-XENIX Development System, you can install the host language capabilities of Unify, which allow you to develop custom database applications. The amount of storage space required of each component is shown for both the Runtime and Development Systems in tape blocks of 512 characters.

The columns under DISK CONFIGURATIONS list the mounted file systems available for installing Unify. For each mounted file system, the amount of available space is shown. A number has been assigned to each file system, which may be used to designate the placement of various Unify components.

If there is not enough space on any mounted file system to install Unify, the following message appears on the bottom of the screen:

> There is not enough disk space to install Unify. You must remove some disk files before trying the installation again. Hit the <ENTER> key to exit.

If there is only enough space to install the Runtime System, the following message is displayed:

> There is only enough space to install the runtime system. Do you want to proceed?

To proceed, type **y**. If you need to remove some files to make room for the Development System, press **n**.

7. If there is enough space available to install the required components of Unify, the following prompt appears on the bottom of the screen:

```
Do you want to install the runtime system or the
development system?
Enter "r" for runtime, or "d" for development:_
```

If you have purchased the TRS-XENIX Development System, type d; otherwise, type r and press <ENTER>.

8. Asterisks (*) appear next to either the Runtime or Development components to indicate the option selected. The following prompt appears on the bottom of the screen:

```
On which disk would you like to install the UNIFY
programs?
Please enter a disk number (or "q" to quit):_
```

9. Select the mounted file system, from those shown under DISK CONFIGURATIONS, on which you want to place the Unify programs.

If there is not enough space on the file system selected, this message is displayed:

```
There is not enough room on that disk.
```

If enough space does exist, the free space required for the programs is subtracted from Free Space Available -- Now, and the result is placed in the Free Space Available -- After Installation column.

1Ø. The file system for the Unify libraries can be selected in the same manner as described above. The libraries can be placed on any mounted file system; they do not have to be on the same file system as the programs.

11. If there is enough disk space remaining to install the tutorial, the following prompt appears:

```
Do you want to install the tutorial?
```

If there is not enough space remaining, this prompt appears:

```
There is not enough space to install the tutorial.
```

If you respond y to install the tutorial but there is not enough space remaining on the mounted file systems to create the database, the following warning appears:

If you install the tutorial, there will not be enough space for an application.

This message gives you an opportunity to either terminate the installation of the tutorial or continue with the installation and remove files to make room for creating a database before executing Unify.

12. Once the Unify programs and libraries have been assigned to a file system and you have made a decision whether to install the tutorial, this message is displayed:

The UNIFY runtime [or development] system will be installed as indicated. Proceed?_

You can now proceed or reassign the placement of the various Unify components on your system. If you answer **n** to the "Proceed?" prompt, the Disk # and Free Space Available After Installation columns are cleared, and the installation program returns to the the following prompt:

Do you want to install the runtime system or the development system?

If you answer y to the "Proceed?" prompt, any previous version of Unify is removed, and the configuration you just specified is installed.

A new screen appears with the following messages:

The runtime [development] system will be installed
Reading first diskette

The screen scrolls through the files being installed, and at the end of the installation of each diskette, the screen asks you to insert the next diskette.

If you are installing the runtime system, the following message appears:

The sixth diskette will not be needed, as it contains the development programs and libraries, and you are

installing the runtime system.

After a short pause, the screen shows:

The UNIFY diskettes have been read successfully.

Installation complete - Remove the diskettes, then press <ENTER>_

Press **<ENTER>** to return to the Installation Menu. The screen shows:

1. to install
q. to quit

Press **q** to quit and return to TRS-XENIX.

13. Remain logged in as root and change to the directory in which you have installed the Unify programs. The Unify installation process places the various components of Unify within the specified file system under the directory structure appl/unify. If, for example, you have indicated that the Unify programs be installed on the root file system (1), the required entry would be:

cd /appl/unify <ENTER>

14. Once you have positioned yourself at the Unify directory, type:

unify <ENTER>

The screen displays the program name, then clears and displays the following message:

Do you wish to create a new data dictionary?_

Type:

y <ENTER>

The following message then appears:

Creating new data dictionary...

An empty data dictionary file (unify.db) is copied from the unify/lib directory, then the Main Menu appears.

Note: Your system displays today's date.

```
+-------------------------------------------------------------+
| [entmenu                      UNIFY SYSTEM                  |
|                          5 OCT 1982 - 15:25                 |
|                               Main Menu                     |
|                                                             |
|  1.  System Menu                                            |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|  SELECTION: _                                               |
|                                                             |
+-------------------------------------------------------------+1
```

15. Unify provides program level security which initially permits access to root only. Once you execute Unify, root should grant 'super user' access for all aspects of Unify to a user through System Parameter Maintenance.

To grant this super user access, type the following at the SELECTION prompt on the Main Menu:

parmnt <ENTER>

The System Parameter Maintenance screen is displayed:

```
[parmnt]                        UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                          System Parameter Maintenance

SUPER USER ID   : su

ENTRY POINT     : entmenu

SYSTEM HEADING  : UNIFY SYSTEM

LANGUAGE        : EN

MONTH MNEMONICS: JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC

BLOCKS / VOLUME: 849
```

Beside the SUPER USER ID prompt, type the three or four character login ID of the user to which Unify 'super user' access is to be granted. After typing in this user ID, be sure to press <ENTER> to commit the change, then exit Unify by pressing <F1> until the TRS-XENIX prompt appears.

16. Verify that you remain positioned in the directory structure appl/unify by typing:

pwd <ENTER>

Then type:

umove <ENTER>

The umove script will copy the data dictionary (unify.db), in which you have just defined a new Unify 'super user', back into the unify/lib directory.

If you inadvertently execute umove from a directory other than appl/unify, the following error message appears:

umove: not found

If you attempt to run umove as any other user than root or at any other terminal than the console, this error message appears:

umove must be run by root from console

If you fail to execute Unify before running umove, the following error message is displayed:

Please run unify before executing umove

When the umove script has been successfully completed, the following message appears:

unify.db has been moved to the unify/lib directory

17. At this time, if you have not yet created the user to which you have granted access to all aspects of Unify, you may do so by executing mkuser. (See the TRS-XENIX Operations Guide for instructions on making a user.) Once you have created the user to which Unify 'super user' access has been granted, login as that user. You are now ready to create a database.

18. Unify databases may be created in any directory you desire. Remaining logged in as your new Unify 'super user', you may change to the directory in which you want to create your database by entering cd followed by the path to the desired directory. We recommend that you do not create your database in the Unify system directory structure, which contains your Unify programs.

Once you position yourself in the directory in which you want to create your database, you must run ucreate. This utility

automatically creates the directory structure required by Unify and sets the appropriate permissions on these directories.

To run the ucreate utility, type the following at the TRS-XENIX prompt:

ucreate <ENTER>

The following options appear on the screen:

```
Make directories/Set permission Unify Database
----------------------------------------------

  1  to make the directories and set the permission
  q  to quit

Please Select:
```

Enter 1 at the selection prompt. The screen then displays the following message:

```
/xx/xx/... is the current directory. Make Unify
directories here? [y,n]
```

Press y to begin the ucreate process. If you press n, you return to the TRS-XENIX prompt. After pressing y, the following messages appear:

```
Making Directories
[list of directories as created]
Setting Permission
```

Note: If you try to create the directories and any of them already exist, ucreate lists the existing directories and creates the others.

Once your application or user directories are created and the permission set, you return to the first screen. Now press q to exit the ucreate utility.

You have completed the initial installation instructions. Now whenever you execute Unify, make sure you cd to the directory in which you created the Unify directory structure, and then type:

unify <ENTER>

The screen clears and the following question appears at the bottom:

Do you wish to create a new data dictionary?_

Unify only asks this question if it doesn't find an existing data dictionary in the directory. Since we want to copy the data dictionary with the new 'super user' access to begin Unify, type:

y <ENTER>

The screen then displays the following message:

Creating new data dictionary..._

When the user's data dictionary file (unify.db) is copied from the system lib directory, the Main Menu reappears.

FILES

Unify uses file-naming conventions to identify the different types of files necessary for running an application. File names have the form name.suffix. Both the name and suffix are of variable lengths, the suffix indicating the type of the file.

Warning: In order to take advantage of the multi-user capabilities of Unify, all files created when executing unify must be owned by a TRS-XENIX user other than root. To create the appropriate multi-user environment, a TRS-XENIX user must be granted 'super user' access to all aspects of Unify according to the installation instructions.

Following is a list of each Unify file name suffix, its meaning, and descriptions of any significant files of its type. Unify files are usually located in a particular directory, noted in the description of the suffix. "Directories" discusses the directory layout in more detail.

.a — An archive file, created with the TRS-XENIX command ar(1), that contains relocatable binary files in a form suitable for loading with the ld(1) command. Some Unify system archives are in the unify/lib directory. The applications developer also creates archive files, usually in a src subdirectory.

.c — A C source code file. This type of file is created by the applications developer and resides in a src subdirectory.

.db — A database file. Every application contains two database files -- the data dictionary, unify.db, and the application database file, file.db. unify.db is copied from the original diskette when you install Unify. file.db is created with Create Database. The prototype data dictionary is located in the unify/lib directory, and your database files are located in the unify/bin directory.

.dbr — The "raw" database file, file.dbr. It is created by Create Database. Every application contains at least one .dbr file. It is either linked to file.db if an ordinary TRS-XENIX file is being used for the database, or it points to the same raw disk extent as file.db if a file system is being used for the database. This is just another way of accessing the application database file.

.h — An "include" file used in conjunction with C source code. This type of file is located in either the def or include directory. Unify creates two kinds of .h files. The first is file.h, the result of a Create

or Reconfigure Database. file.h contains a list of the record and field names for use by C programs that access the database. The second type of .h file is screen.h, created by Process Screen. This type of file contains a list of the screen field names, again for use by C programs using that particular screen form.

.idx — A B-tree index file. These binary files reside in the unify/bin directory. They are created by Secondary Index Maintenance and are updated by any program that stores data in an indexed field. The files named btnnnnn.idx are actual B-tree files; the field.idx file is a directory to the index files.

.o — A relocatable binary file resulting from a cc -c. This type of file is usually kept in an archive file.

.q — A Unify screen form file. This type of file contains binary versions of screen forms that can be read quickly at run time. .q files are produced by Process Screen. Unify system screen forms are kept in the unify/lib directory; your screen forms are usually stored in the unify/bin directory.

DIRECTORIES

Unify uses two basic types of directories. **System** directories are used by all Unify users and should **never** be modified. These directories are included on your Unify installation diskettes. Each application has its own set of **application** or **user** directories that contain the files specific to a particular application. You may set parameters within /usr/bin/unify so that both types of directories can be located wherever convenient. The Unify system directories are laid out as follows:

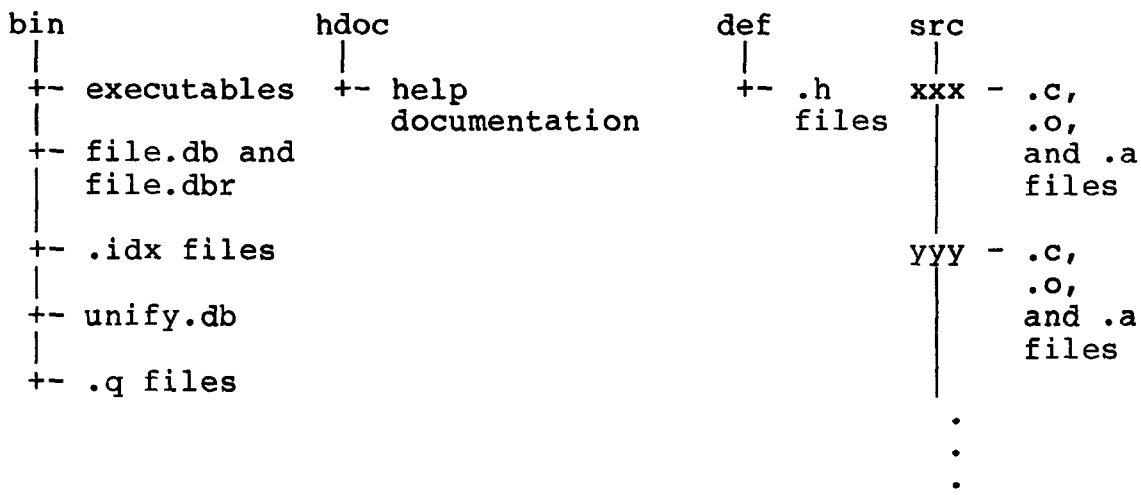
```
bin              lib              include
 |                |                  |
 +-system         +- system          +- system
   executables    |  .a files           .h files
                  |
                  +- system
                  |  .q files
                  |
                  +- prototype
                  |  unify.db
                  |
                  hdoc -+
                        |
                        +- system help
                           documentation
```

Only the bin and lib directories are used at run time. The files in **include** are used when compiling programs.

In addition to the system directories, you also receive a sample application already developed in the tutorial directory. This application directory is laid out according to Unify directory conventions, discussed in "UNITRIEVE Database Maintenance." The tutorial directory is designed as follows:

```
                      tutorial
                         |
bin ------------- def -+----- src ------------------- hdoc
 |                 |           |                       |
                                                       +-.n
 +- executables    +- .h     ord - Order Maintenance     files
 |                    files  |     .c, .o, and .a files
 +- file.db and              |
 |  file.dbr                 sfe - Custom ENTER
 |                                 .c, .o, and .a files
 +- unify.db
 |
 +- .q files
 |
prep - text for runutut
```

A typical application or user directory structure, which can be created by running ucreate from the TRS-XENIX or Unify shell, would look like the following diagram. In the diagram, xxx and yyy represent src subdirectories.



Warning: If you create these directories without using the ucreate utility, you must set read, write, and execute permissions for both the owner and group to let multiple users access Unify.

MULTIPLE DATABASES

With Unify, you can include many applications in a single database by designing menus and assigning access privileges to individual users to limit their view of the database to specific applications. But you are not limited to one database. As an alternative, you can design multiple databases, providing each database is located in a different directory.

To create and use a new database in another directory, first change to the directory desired and execute ucreate. ucreate will make a directory structure as shown in the previous section, "Directories," for designing your new database. Now execute Unify to create your new database in the bin subdirectory of the directory you have selected. The unify script contained in /usr/bin has parameters set which automatically locate the appropriate Unify programs and libraries required to design your application.

In order to define help documentation for your application and to utilize the host language capabilities of Unify, some additional subdirectories have been created in the selected directory by ucreate. Unify locates the database help documentation, archives, and define files in the other subdirectories by changing the directory back to the "root" level of your directory structure and then changing to the subdirectory required.

If you have already created a database elsewhere on your system that you want to copy to your new directory structure, be sure to copy the files to the appropriate subdirectory. Then **cd** to the bin subdirectory, remove the file.dbr, and relink it with file.db by entering **ln file.db file.dbr**. You are now ready to execute Unify.

Warning: You will get erroneous results if the permissions on the Unify subdirectories are modified or the sub-directories are not kept at the same level of the directory structure. To preserve the multi-user capabilities of Unify, it is necessary that a TRS-XENIX user other than root be given 'super user' access to all aspects of Unify according to the installation instructions.

PARAMETERS AND ENVIRONMENT VARIABLES

Unify provides a convenient way of setting up parameters that Unify programs can access. These parameters, defined within

the script /usr/bin/unify, define the directory paths used by Unify Programs to locate files. Parameters take the form name=value. Both name and value are arbitrary strings.

An example of changing parameters is using a directory to find a specific executable file. The value of the PATH parameter tells the shell which directories to search when looking for a specific executable file. The default value for PATH is :/bin:/etc:/usr/bin, so normally the shell looks in the directories /bin, /etc, and /usr/bin. To put Unify executable files into appl/unify/bin, the unify script simply adds this directory to the PATH search list. To modify the default search list defined within the script /usr/bin/unify, use a text editor to change the value of the parameter PATH.

You can also set parameters and export them in your .profile file. They are then automatically set when you log in. All variables must be exported to work properly.

Following is a list of Unify parameters you may set. These variables let you change the standard directories for executable programs and other supporting files.

PATH — Must contain the directory in which the Unify system executable files are located. It can also contain the directory in which your application's executable files are located if you want to execute them from somewhere other than the current directory. The standard TRS-XENIX PATH is set to :/bin:/etc:/usr/bin. The unify script adds the subdirectories appl/unify/bin to the search list preceded by the name of the mounted file system on which you installed Unify. If you put your Unify executable files in any of these places, you need not change anything.

UNIFY — The location of the Unify lib directory. The path of the Unify lib directory is set by default to the sub-directory appl/unify/lib within the mounted file system you specified at installation. You may change the search list of the lib directory by modifying the

value of the parameter **unify** in the script /usr/bin/unify.

TERM — The type of your terminal, as defined in your termcap file. The unify script examines the ttytype file to determine the type of terminal from which Unify was executed. There should be no need to change the terminal type.

TERMCAP — The path name of the termcap file. The default for this is appl/unify/lib/termcap. If a change is made to the search list for the parameter **unify**, it affects the path to the Unify termcap.

DBPATH — The directory in which your Unify system data files are located. This includes the file.db, file.dbr, and unify.db files, your .idx files, and your .q files. Unify also expects to find any user help documentation in the $DBPATH/../hdoc directory. If you set this variable, **all** the above files must be located in the indicated directories to be found. By setting DBPATH, PATH, and UNIFY correctly, you can be in any directory and still execute your Unify application. By default, DBPATH is not set; all above files are assumed to be in the current directory (or in the case of help documentation, in ../hdoc).

The installation procedure places the data dictionary, unify.db, in the search list appl/unify/bin on the mounted file system of your choice. To run Unify, you can: (1) set appl/unify/bin as your current directory, then execute Unify or (2) execute Unify from another directory and create a new data dictionary. For more information, see the section on Multiple Databases in this Introduction.

BUDEV — The name of the device to/from which you want to backup and restore the database. By default, this parameter is set in the unify script to floppy diskette Ø (/dev/rfdØ).

EDIT The name of your favorite text editor or word processor. The default is the vi editor, but if you prefer another one, you may change it. The editor is used by Enter Help Documentation and by SQL when editing queries.

TERMCAP

The hardware functions of most CRT terminals can be described by the Unify termcap file that resides in appl/unify/lib. The format and use of this file is described in the TRS-XENIX System Reference Manual, in termcap(5). Unify uses this file to obtain information on performing cursor addressing, screen control, and keyboard input mapping.

Unify can function using just four of the standard terminal capabilities. However, Unify can also use some unique entries not described. You may use these entries to customize the usage of the input keys on your terminal as well as the output characteristics of your terminal. The following tables explain the Unify termcap entries and the ways in which they are used.

Required Terminal Functions

cd	Clear to end of display
ce	Clear to end of line
cl	Clear all
cm	Cursor motion

The above characteristics are required for using Unify. The termcap file provided with Unify for Models II/12/16 and the DT-1 contains these required functions.

Optional Terminal Functions

cf	Clear unprotected fields
ps	Start protect mode
pe	End protect mode
ws	Start write protect (half intensity)
we	End write protect (restore full intensity)
so	Begin stand out mode (reverse video or similar)
se	End stand out mode
us	Start underline
ue	End underline

The first five entries are grouped together, since these terminal functions do not perform properly if some of the characteristics are included and others are omitted. They are optional, but only as a group.

The final four output characteristics are also optional. If you like, your Model II/12/16 can display menu lines and data entry prompts on SFORM screens in reverse video. The DT-1 supports underline mode as well as reverse video. Refer to the sections that explain Executable Maintenance, Screen Maintenance, and ENTER Screen Registration for information about using these features. Basically, the escape sequences, ~r and ~s in prompts and menu lines, turn reverse video on and off, while ~u and ~v turn underline on and off.

Keyboard Mapping

Es	Begin search (using ENTER screen); default is <CTRL><E>
Ec	Clear entry (used by ENTER); default is <CTRL><Z>
Ex	Exit ENTER; default is <CTRL><X>
Uf	Move forward/downward in screen/menus; default is <ENTER>
kØ	Move backward/upward in screen/menus; default is <F1>
kl	Accept entry; default is <F2>

These entries let you substitute different control characters for the standards used by Unify in the menu handler and on ENTER screens. You may customize these control characters as you like. The termcap file entries described must be in the form name=^X. name stands for the two-character code of a characteristic, and X is an uppercase letter other than H, J, and M.

The kØ control code assigns a value for the control key to return to the previous field or prompt. The default is <F1>, but you may change this parameter to whatever you like. For example, if kØ is set to ^P (kØ=^P) in the termcap file, <CTRL><P> acts like <F1> when you use Unify screen forms and menus.

The kl control code assigns a value for the control key to store an entry in the database. The default is <F2>, but you may change this parameter to utilize a different key.

In the Unify termcap, no entries have been made for the control codes Uf,Es,Ec, and Ex. The defaults for these controls are as defined below. Should you want to modify these control keys, simply add the control code and the value of the new key sequence you wish to utilize.

Es, Ec, and Ex are used to set control keys for ENTER. While using ENTER, a message tells you which keys are understood. The defaults for search, clear, and exit are <CTRL><E>, <CTRL><Z>, and <CTRL><X>.

A Uf entry in the termcap file indicates that another control key can perform the same function as <ENTER> does in Unify.

Setup

If you wish to make changes to the termcap, we recommend that you create a termcap file containing only entries for your terminal types. Be sure the TERMCAP pathname defined within the unify script, /usr/bin/unify, searches for your termcap in the appropriate directory.

If you have made any changes to the names of the terminal types in the termcap file, you must set the TERM environment variable to your type of terminal and set the terminal speed. Using the TRS-XENIX shell, this is done by entering:

```
TERM=xxxx
export TERM
stty nnnn
```

xxxx is the code for your terminal type in the termcap file, and nnnn is the baud rate at which you are running (96ØØ). Both these parameters may be set in your .profile. This way, you do not have to set parameters every time you log in.

MENUH -- MENU HANDLER

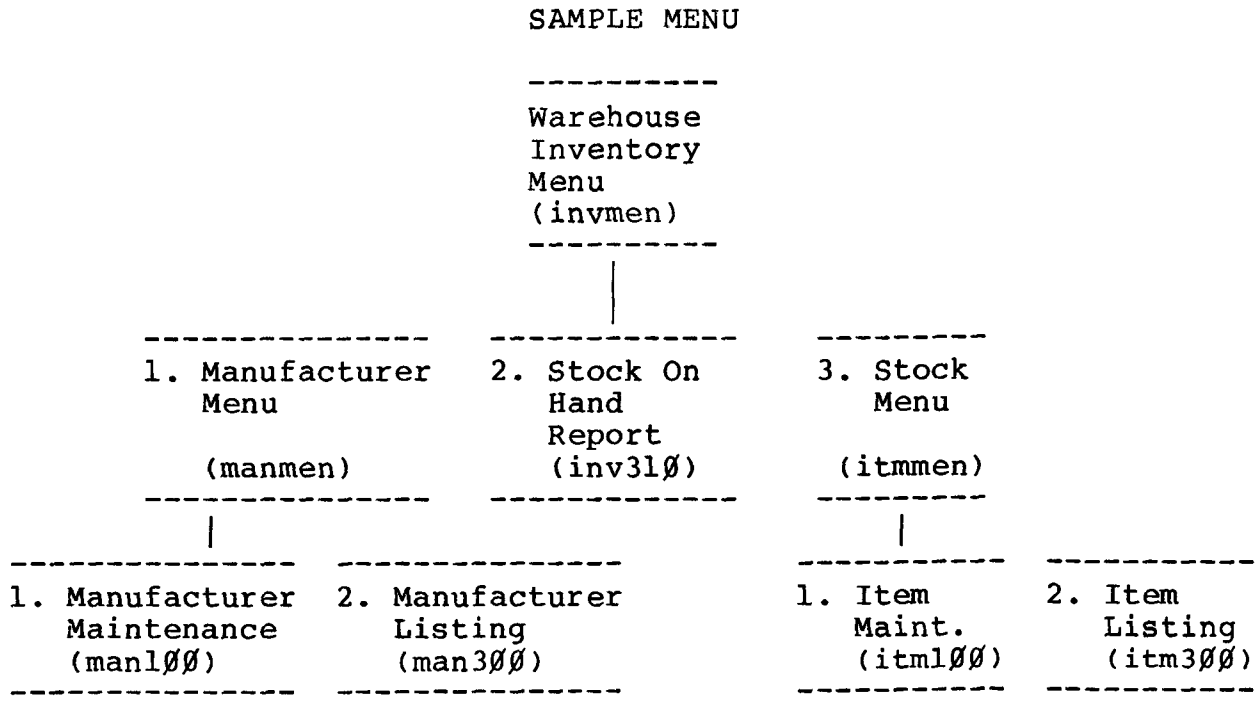
The menu handler and security system, MENUH, gives you a complete, user-friendly environment for the Unify system. Use it when you develop a project, when you use Unify to create an application, and when you operate the finished application.

MENUH controls all your interactions with Unify. It updates the data dictionary to keep track of the menus, application functions, ENTER screens, and users in an applications system. MENUH also finds the archives for each application program. It determines which programs are loaded together in a single executable core load and which SFORM screen to display when a program begins execution.

The initial data dictionary contains only the Unify programs and menus and permission settings for /usr/root, the Unify "super user." You can use MENUH during the development of an application to create the menu structure for your system.

Menus

A menu is a set of selections from which you may choose. Each selection represents either a function, a program, or another menu. MENUH allows a set of disjoint, hierarchical menu trees to be created and maintained. Following is a sample menu/program tree for a subset of a warehouse inventory system.



The primary menu gives you the following options:

1. Manufacturer Menu
2. Stock On Hand Report
3. Stock Menu

You have three options. If you press 1, the Manufacturer Menu is displayed:

1. Manufacturer Maintenance
2. Manufacturer Listing

To return to the previous menu, press <F1>. If there is no previous menu, control is returned to the shell. In this case, <F1> would redisplay the Warehouse Inventory Menu.

If you enter 2 at the Warehouse Inventory Menu, the Stock On Hand Report program is run. When the program finishes executing, the Warehouse Inventory Menu is redisplayed. Enter 3, and the Stock Menu is displayed:

1. Item Maintenance
2. Item Listing

In addition to moving through the menus in the above manner, you can execute programs and menus directly. Assign each menu and program a name. This name is always displayed in the upper left corner of the screen when that menu or program is running. Programs and menus can then be executed by typing their names instead of pressing a number key. For example, you could go directly to the Manufacturer Maintenance program by typing **manlØØ** <ENTER>.

Users

To access Unify, you must login as a TRS-XENIX user and position yourself at the database directory structure. From your user ID, MENUH determines which menu to display first and which options on this and subsequent menus are available for your use. When a program is executed, MENUH passes that program the current user's access privilege. You may display only the menus and programs that match your access privileges.

Each user has a defined access privilege to every program and menu. You may have either no access (the default condition), or you may have access to inquire, modify, delete, and/or add. The four access capabilities may be freely mixed. As an example, you might have inquire and delete privileges in a program, modify and add privileges, inquire only privileges, and so on. This privilege is passed as a number interpreted by the application program, which then allows only the indicated operations. The utility function, **priamd**, is included in the Unify applications library for this purpose.

Beside access privileges, you also define an entry menu for each user. An entry menu is the first menu seen after logging into Unify. You may see only the portion of the menu tree that applies to you. Since some programs might appear on multiple menus, the system administrator can tailor each user's entry menu and privileges exactly as required.

For convenience, each user is assigned to a group. A group is a collection of users with similar access privileges. You can assign a default set of privileges to the group and then enter only the differences for each user. For example, data entry clerks could be defined as a group because they have access to the same programs. Warehouse employees would have access to a different set of programs. A new employee might have only inquire privileges.

The Unify system requires a special "super user." This user has automatic access to all menus and programs. The super user id and entry point are all defined in System Parameter Maintenance, discussed later in this section. The default for

the id is **su** (which stands for the user root). The entry menu defaults to entmenu.

Help Documentation

MENUH associates help documentation with every program and menu registered with Executable Maintenance. This documentation is created by using the Enter Help Documentation function. At the SELECTION prompt of a menu, type **help** <ENTER> to display the help documentation associated with the current screen. You may also type **help n** <ENTER> or **help xxxx** <ENTER>, where **n** is a valid selection number or **xxxx** is a menu or program name. This displays the help documentation associated with the menu or program you specified.

Executables

DISREGARD THIS SECTION UNLESS OPERATING WITH THE TRS-XENIX DEVELOPMENT SYSTEM.

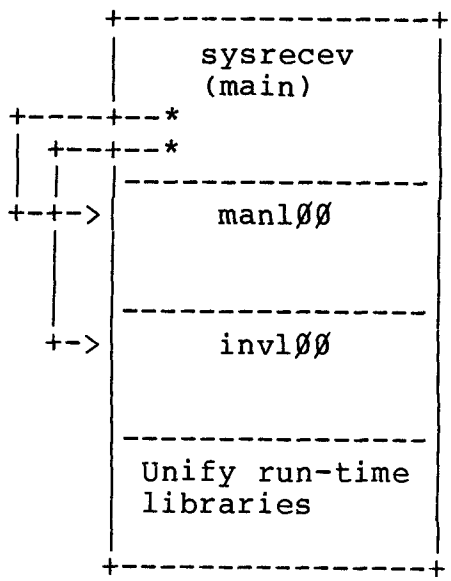
An "executable" is either a TRS-XENIX shell script or a collection of compiled source language programs loaded in a single TRS-XENIX archive. If the executable is an archive file, each of the programs must have an entry point (a C function) with the same name as its MENUH registered name.

The main routine in a Unify archive executable is **sysrecev**. The sysrecev function accepts the arguments from the menu handler, sets up the terminal type, performs other housekeeping tasks, and then calls the correct program. Loading several programs into one archive saves both diskette and memory space.

If the executable consists of only one application program, sysrecev is not necessary. The menu handler simply executes the program, and that program is responsible for establishing its own environment.

Consider the sample executable file, INV1ØØ, in the following illustration. It contains two applications programs, man1ØØ (Manufacturer Maintenance) and inv1ØØ (Inventory Maintenance). sysrecev is used as the main routine, and the Unify run time libraries are also included.

UNIFY EXECUTABLE FILE - INV1ØØ



Programs

DISREGARD THIS SECTION UNLESS OPERATING WITH THE TRS-XENIX DEVELOPMENT SYSTEM.

Each program in an executable must be registered with Unify through MENUH. You must specify the prompt to appear on every menu in which the program is used. This prompt also appears at the top of the screen when the menu handler executes the program. Optionally, you may specify the SFORM screen to be displayed before the program begins execution. You may also specify the **src** subdirectory at which the program's object code archive is located. The latter is used by Program Loading.

Program Execution via the Menu Handler

When you select a program from a menu, the menu handler looks in the data dictionary to see which executable file contains that program. MENUH then starts the executable by using the TRS-XENIX system call, execvp(2). If the program is in an executable that does not use sysrecev, the menu handler passes no parameters to the executable -- it simply starts it. Executables like this can contain only a single program, unless you have coded a method (other than examining a parameter list) of determining which program to start.

If the program is in an executable that uses sysrecev, the menu handler constructs a parameter list containing the following elements:

ENAME T <index> <access level> <language code> <screen name>

The parameters have the following meanings:

Parameter	Usage	Source
ENAME	Name of the executable	Executable Maintenance
T	Not used, present for historical reasons	Constant value
<index>	The entry number within the executable of the particular program to start	Executable Maintenance
<access level>	Integer from Ø to 15 that defines whether this user has ADD, INQUIRE, MODIFY, or DELETE privileges	Group Maintenance or Employee Maintenance

Parameter	Usage	Source
<language code>	System language code	System Parameter Maintenance
<screen name>	Name of the screen form associated with this program	Executable Maintenance

Consider the INV1ØØ executable file pictured previously. It would be registered using Executable Maintenance as executable name INV1ØØ, using sysrecev, with two programs: man1ØØ at entry Ø, using screen sman1ØØ, and inv1ØØ at entry 1, using screen sinv1ØØ. If you selected man1ØØ (Manufacturer Maintenance) from a menu when logged in as the Unify super user, the following parameter list would be passed to INV1ØØ:

```
INV1ØØ T Ø 15 EN sman1ØØ
```

This assumes that the standard language code for English (EN) has been left unchanged. If you selected invlØØ (Inventory Maintenance) from a menu, the following parameters would be passed to INVlØØ:

```
INVlØØ T 1 15 EN sinvlØØ
```

When INVlØØ starts executing, sysrecev runs first, since it is the main function. It first puts the parameters into global variables as follows:

Parameter	Variable
T	Not saved
<index>	Not saved
<access level>	int logacl
<language code>	char langtp[2]
<screen name>	char *_savscr

It then uses the <index> to execute the appropriate user function defined as the program entry point. This works by using a table of pointers to functions previously built by Program Loading. The table's name is menucall, and it is defined in the source file prgtab.c. The source code for the example executable would appear as follows:

```
int manlØØ(), invlØØ();
int (*menucall[])() = {
                        manlØØ,
                        invlØØ
                      };
```

You can see from this table definition that manlØØ is at entry Ø, while invlØØ is at entry 1.

Using unify as the Shell

The menu handler consists of three programs that perform all the functions described in this section. The startup program is unify. Type **unify** <ENTER> at the shell to begin executing a Unify application. **unify** is the program that actually displays the menus and controls security. The executable, MENUH, contains all the programs described in "Menu Maintenance Screens." The executable, RMENU, contains the programs described in "Menu Handler Reports."

If you want, you may use unify as the shell for TRS-XENIX users. When users log in to TRS-XENIX, Unify will execute automatically and display the first Unify Menu (instead of the TRS-XENIX shell prompt). To do this, you should be familiar with the format and usage of the TRS-XENIX password and .profile files, described in the TRS-XENIX System Reference Manual in login(1), passwd(1), and passwd(5). You should also be familiar with the Unify parameters and environment variables discussed in the first section of this manual.

To use unify as the shell for a user, you must set up the proper entry in the TRS-XENIX password file and set up the .profile file to establish the appropriate environment and execute unify. An entry in the password file gives the home directory for the user. If you are not using the DBPATH environment variable, the Unify data dictionary must be in the user's home directory.

Suppose, for example, that you want TRS-XENIX user "john" to log directly into the tutorial application. Set up the TRS-XENIX user id as the Unify user id within Employee Maintenance. John's user id for both TRS-XENIX and Unify is then "john." Assume that the directory structure is the same as described in the first section of the manual and that tutorial is located under appl/unify on the root file system. To specify the user's home directory, the appropriate password file entry would be:

john::nn:nn::/usr/appl/unify/tutorial/bin: <ENTER>

Note that the shell specification in the password file entry should be left open so that the default shell executes the .profile file in the home directory. The .profile file must be modified to include the following commands, assuming that the Unify system files are in the locations specified:

```
UNIFY=/appl/unify/lib
PATH=:/bin:/usr/bin:/etc:/appl/unify/bin
export PATH UNIFY
unify
kill -9 $$
```

The last command in the profile is kill -9 $$, which returns the user to the TRS-XENIX login after exiting Unify.

With this file, any user that logs into the tutorial directory as home and has the same Unify and TRS-XENIX user id,sees the appropriate Unify menu displayed immediately after logging in to TRS-XENIX. Users without a valid Unify user id would not be permitted to log in at all.

MENUH Maintenance Screens

This section describes the screen entry programs used to maintain executables, menus, groups, employees, help documentation, and system parameters in the data dictionary. It also explains how to load binary executable files using the standard Unify load macro.

EXECUTABLE MAINTENANCE

DISREGARD THIS SECTION UNLESS OPERATING WITH THE TRS-XENIX DEVELOPMENT SYSTEM.

Executable Maintenance lets you add, modify, and delete executable files. You can also specify the programs contained in a given executable. After you select Executable Maintenance (execmnt) from the MENUH Screen Menu, the screen shows:

```
[execmnt]                          UNIFY SYSTEM
                                 5 OCT 1982 - 15:25
                               Executable Maintenance

EXECUTABLE'S NAME: ''''''''
USES SYSRECEV ? '
PROGRAMS:
cmd  NAME        HEADING                                SCREEN     DIRECTORY
  Ø  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  1  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  2  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  3  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  4  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  5  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  6  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  7  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  8  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""
  9  """""""""   """""""""""""""""""""""""""""""""""""  """""""""  """""""""""""""""""""



[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE _
```

```
''''''''' = executable entry area
""""""""" = program entry area
```

This screen lets you inquire, add, modify, and delete executables. An executable may contain multiple entry points, each of which may start up a different program. This usually applies to custom programs, not command files. Following is an explanation of each prompt on the screen.

[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE

Lets you choose an operational mode for the program. The modes are:

i -- Inquire mode, which lets you see an executable already registered with MENUH.

a -- Add mode, which lets you register a new executable with MENUH.

m -- Modify mode, which lets you modify HEADING, SCREEN, and DIRECTORY for programs within an existing executable or add and delete programs.

d -- Delete mode, which lets you delete an existing executable and all its associated programs.

EXECUTABLE'S NAME

The eight-character name of the TRS-XENIX file containing the executable. This filename is passed to the shell when a program within the executable is selected.

USES SYSRECEV ?

Answering y means that the executable is passed the standard argument list from the menu handler. n means that no arguments are passed when this executable is started. The sysrecev function usually interprets this argument list, but any program or shell script can request that these arguments be passed.

CMD

Used for entering a command to perform an operation on the current program. The cursor moves up and down in the multi-line entry area of the screen in response to <F1> and <ENTER>. The cursor position marks the current program. Valid commands are:

a -- Add a new program to the executable. This response is valid only on the first empty line in the multi-line area. In other words, you must move the cursor to the first blank line to add a new program. Once a program is added, press <F2> to store it.

m -- Modify HEADING, SCREEN, or DIRECTORY for the current program. To modify NAME, you must delete the line and re-add it.

d -- Delete the current program.

q -- Leave the multi-line entry area, and position the cursor at the USES SYSRECEV ? prompt.

NAME

Eight-character name of the program within the executable. Type this name in response to the SELECTION prompt to execute the program. If the executable is a custom program and the name is an entry point into the executable, this name must be the same as the host language (C, RMCOBOL) function name that is the entry point for the program.

HEADING

A 36-character string displayed on the third line of the screen when the program is running and on menus in which the program is a selection. You can enter literal text, or "escaped" characters, indicating that the prompt is to be displayed in a special mode. The escape character is ~. Models II/12/16 support these special modes:

~r -- start reverse video
~s -- end reverse video

The DT-1 supports the following modes in addition to those above:

~u -- start underline
~v -- end underline

If you start a special display mode for a heading, be sure to end it, or your screen form will not seem proper. Reverse video or underline is only displayed on menus, not on the third line when the program is running.

SCREEN

The name of an SFORM screen to be displayed by sysrecev before the program is started. This entry is optional.

DIRECTORY

For custom programs, this is the TRS-XENIX directory in which the source code for the program is found. The Unify program loading macro (uld) expects the archive containing the program to be in this directory as well. The relative path ../src/ is prefixed to the directory name entered here. If you leave this field blank, uld uses the first three characters of the program name for the directory name. For example, if the program name is invlØØ, uld would look in ../src/inv for the archive containing invlØØ.

If you make any change or addition in Executable Maintenance to a program that appears as an option on a menu, return to the menu **prior** to the menu on which this program appears as an option, and then reselect the menu containing the program. This procedure is necessary to reflect the changes you made in Executable Maintenance.

MENU MAINTENANCE

Menu Maintenance lets you add, modify, and delete menus in the data dictionary. After you select Menu Maintenance (menumnt) from the MENUH Screen Menu, the screen displays:

```
+-------------------------------------------------------------------------+
|[menumnt]                    UNIFY SYSTEM                                |
|                          5 OCT 1982 - 15:25                             |
|                           Menu Maintenance                              |
|                                                                         |
|NAME: ######## HEADING: ###################################              |
|CMD  LINE  MENU/PROG   M/P   PROMPT                                      |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
| "     ""   """"""""    "    """""""""""""""""""""""""""""""""""         |
|[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE _                                  |
|                                                                         |
+-------------------------------------------------------------------------+
```

######## = menu entry area
"""""""" = menu line entry area

This screen lets you inquire, add, modify, and delete menus and menu lines. There is an entry area for menus and a multi-line entry area for menu lines. Following is an explanation of each prompt and column heading on the screen.

[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE

Lets you choose a mode for the program. The modes are:

i -- Inquire mode, which lets you see an existing menu.
a -- Add mode, which lets you create new menus.
m -- Modify mode, which lets you add, modify, and delete lines on an existing menu.
d -- Delete mode, which lets you delete an entire menu.

NAME

An eight-character menu name. The name must be distinct from other menu, screen form, and program names.

HEADING

A 34-character string to be displayed on the third line of the screen when the menu is displayed and on other menus where the menu is a selection. You may enter literal text, or "escaped" characters, indicating that the prompt is to be displayed in a special mode. The escape character is ~. Models II/12/16 support these special modes:

~r -- start reverse video
~s -- end reverse video

The DT-1 supports the following modes in addition to those above:

~u -- start underline
~v -- end underline

If you start a special display mode for a heading, be

sure to end it, or your screen form will not seem proper. Reverse video or underline is displayed only on the menu, not on the third line when the menu is running.

CMD

Used for entering a command to perform an operation on the current menu line. The cursor moves up and down in the multi-line entry area in response to <F1> and <ENTER>. The cursor position marks the current menu line. Valid commands are:

a -- Add a new line to the menu. This response is valid only on the first empty line in the multi-line area. In other words, you must move the cursor to the first blank line before adding a new menu line. Press <F2> to store a new menu line.

m -- Modify LINE or MENU/PROG for the current menu line.

d -- Delete the current menu line.

q -- Leave the multi-line entry area, and position the cursor at the HEADING prompt.

LINE

The position in the menu that the current menu line should occupy. Any number from 1 through 16 may be entered. All Other menu lines on the screen are shifted accordingly.

MENU/PROG

The name of the menu, program, or ENTER screen to be executed when this menu line is selected.

M/P

A display only field indicating the type of the MENU/PROG entry. The types are:

M -- Indicates that the current menu line starts a menu when selected.

P -- Indicates that the current menu line starts a program when selected.

E -- Indicates that the current menu line starts an ENTER screen when selected.

PROMPT

The text to appear on the menu for the current menu line. For menus, the prompt is entered in this program; for programs, it is entered in Executable Maintenance (execmnt); and for ENTER screens, it is entered in ENTER Screen Registration (entmnt).

If you make any change or addition to a menu in Menu Maintenance, return to the menu prior to the menu changed, and then reselect the menu you have modified. This procedure is necessary to reflect the changes you made in Menu Maintenance.

GROUP MAINTENANCE

Group Maintenance lets you add, modify, and delete group access privilege records in the data dictionary. These records specify which programs and menus a group is allowed to use as well as the level of access allowed. After you select Group Maintenance (grpmnt) from the MENUH Screen Menu, this screen appears:

/

```
+---------------------------------------------------------------------------+
|[grpmnt]                        UNIFY SYSTEM                                |
|                            5 OCT 1982 - 15:25                              |
|                             Group Maintenance                              |
|                                                                            |
|GROUP ID: ####   NAME: ##############################                       |
|                                                                            |
|SYSTEM ENTRY PT: ######## - #                                               |
|                                                                            |
|ACCESS LEVELS:                                                              |
| LN  MENU/PROG M/P INQ ADD MOD DEL  LN  MENU/PROG M/P INQ ADD MOD DEL       |
| ''  ''''''''   '   '   '   '   '                                           |
|                                                                            |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
| ""  """"""""   "   "   "   "   "   ""  """"""""   "   "   "   "   "        |
|                                                                            |
|[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE: _                                     |
|                                                                            |
+---------------------------------------------------------------------------+
```

######## = employee group entry area
'''''''' = access privilege entry area
"""""""" = access privilege paging area

This screen lets you inquire, add, modify, and delete employee groups and access privileges. There are entry areas for employee groups and access privileges as well as a paging area for access privileges. Following is an explanation of each prompt and column heading on the screen.

[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE:

Lets you choose a mode for the employee group entry area. The modes are:

i -- Inquire mode, which lets you see the first page of access privileges for an existing employee group. If more than 2Ø access privileges have been defined for a group, use the modify mode to view subsequent pages.

a -- Add mode, which lets you add new employee groups and associated access privileges.

m -- Modify mode, which lets you add, modify, and delete access privileges for an existing employee group.

d -- Delete mode, which lets you delete an existing employee group and all its access privileges.

GROUP ID

A four-character code identifying the employee group.

NAME

A 3Ø-character employee group name, used for identification and documentation only.

SYSTEM ENTRY PT

The name of the initial menu seen by employees in this group when they log into Unify. In add mode, be sure to define a system entry point.

[N]ext page, [P]rev page, [A]dd line, or number:

Paging area prompt displayed after you answer the SYSTEM ENTRY PT prompt. The paging prompt lets you choose the mode for the paging area. The selections are:

n -- Displays the next page of access privileges.

p -- Displays the previous page of access privileges.

a -- Lets you add new access privileges by moving the cursor to the access privilege entry area. Press <F2> to store a new access privilege.

number (1-999) -- Displays the page containing the indicated access privilege, and positions the cursor at that access privilege to let you modify or delete it.

column to left of LN (untitled)

Area used for entering a command to perform an operation on the current access privilege. It corresponds to the CMD column on the Schema Entry screen. The cursor moves up and down in the paging area in response to <F1> and <ENTER>. The cursor position marks the current line. Valid commands are:

m -- Modify INQ, ADD, MOD, or DEL access privileges if it is for a program or ENTER screen.

d -- Delete the access privileges for this menu, program, or ENTER screen.

q -- Redisplay the paging prompt at the bottom of the screen.

LN

A line number assigned by the system, used to reference the access privilege if you wish to modify it.

MENU/PROG

The name of a menu, program, or ENTER screen to which employees in this group can have access.

M/P

A display only field indicating the type of entry made in the MENU/PROG column. The types are:

M -- Indicates that the current access privilege is for a menu.

P -- Indicates that the current access privilege is for a program.

E -- Indicates that the current access privilege is for an ENTER screen.

INQ

A **y** answer in this column indicates that employees in this group can use Inquire mode for the program or ENTER screen. **n** indicates that they cannot.

ADD

A **y** in this column indicates that employees in this group can use Add mode for the program or ENTER screen. **n** indicates that they cannot.

MOD

A **y** in this column indicates that employees in this group can use Modify mode for the program or ENTER screen. **n** indicates that they cannot.

DEL

A **y** in this column indicates that employees in this group can use Delete mode for the program or ENTER screen. **n** indicates that they cannot.

EMPLOYEE MAINTENANCE

Employee Maintenance lets you maintain employee access privileges. You can add employee privileges or modify/delete existing privileges. You can also maintain a list of discrepancies between an employee's access privileges and the group's access privileges. Where a program or menu access item appears for an employee, it supercedes the group item. When you select Employee Maintenance (empmnt) from the MENUH Screen Menu, your screen shows:

```
+-------------------------------------------------------------------------+
|[empmnt]                          UNIFY SYSTEM                           |
|                                5 OCT 1982 - 15:25                       |
|                              Employee Maintenance                       |
|                                                                         |
|LOGIN ID: #### NAME: ##############################                     |
|GROUP ID: #### NAME: ##############################                     |
|                     SYSTEM ENTRY PT: ########  - #                      |
|                                                                         |
|ACCESS LEVELS DIFFERENT FROM GROUP:                                      |
|  LN  MENU/PROG M/P  ACCESS?  INQ ADD MOD DEL                            |
|  ''  ''''''''   '     '       '   '   '   '                             |
|                                                                         |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|  ""  """""""""  "     "       "   "   "   "                             |
|                                                                         |
|[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE: _                                  |
+-------------------------------------------------------------------------+
```

```
######## = employee entry area
'''''''' = access privilege entry area
"""""""" = access privilege paging area
```

This screen lets you inquire, add, modify, and delete employees and access privileges. Use it to specify differences in this employee's privileges from those in the employee's group. Group privileges determine each employee's basic privileges, but privileges may vary (for each employee) based on what you specify on this screen.

There is an entry area for employees and access privileges as well as a paging area for access privileges. Following is an explanation of each prompt and column heading on the screen.

[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE:

Lets you choose a mode for the employee entry area. The modes are:

i -- Inquire mode, which lets you see the first page of access privileges for an existing employee. If more than 1Ø access privileges have been defined for an employee, use the modify mode to view the subsequent pages.

a -- Add mode, which lets you add new employees and associated access privileges.

m -- Modify mode, which lets you add, modify, and delete access privileges for an existing employee.

d -- Delete mode, which lets you delete existing employees and all their access privileges.

LOGIN ID

A three or four character code identifying the employee. The employee uses this code when he logs into TRS-XENIX and executes Unify.

NAME

A 3Ø-character string for the employee's name, used for documentation and identification only.

GROUP ID

A four-character code identifying the group of which the employee is a member.

NAME

A display only field showing the employee group name.

SYSTEM ENTRY PT

The initial menu seen by the employee when logging in. This field defaults to the system entry point of the group.

[N]ext page, [P]rev page, [A]dd line, or number:

Paging area prompt displayed after you answer the SYSTEM ENTRY PT prompt. The paging prompt lets you choose the mode for the paging area. The selections are:

n -- Displays the next page of access privileges.

p -- Displays the previous page of access privileges.

a -- Lets you add new access privileges by moving the cursor to the access privilege entry area. Press <F2> to store an access privilege.

number (1-999) -- Displays the page containing the indicated access privilege, and positions the cursor at that access privilege to let you modify or delete it.

column to left of LN (untitled)

Area used for entering a command to perform an operation on the current access privilege. It is used in the same way as the corresponding column on the Group Maintenance screen. The cursor moves up and down in the paging area in response to <F1> and

<ENTER>. The cursor position marks the current line. Valid commands are:

m -- Modify ACCESS, INQ, ADD, MOD, or DEL for the current access privilege.

d -- Delete the current access privilege.

q -- Redisplay the paging prompt at the bottom of the screen.

LN

A line number assigned by the system, used to reference the access privilege if you wish to modify it.

MENU/PROG

The name of a menu, program, or ENTER screen to which the employee should have a different access privilege than others in the group.

M/P

A display only field indicating the type of entry made in the MENU/PROG column. The types are:

M -- Indicates that the current access privilege is for a menu.

P -- Indicates that the current access privilege is for a program.

E -- Indicates that the current access privilege is for an ENTER screen.

ACCESS?

A **y** in this column indicates that this employee has access to the indicated menu, program, or ENTER screen. **n** indicates no access. Unless a **y** is entered in this column, the following access modes (INQ, ADD, MOD, DEL) cannot be specified on the menu, program, or ENTER screen.

To access SQL and LST, you must enter y in this column. Subsequent access modes need not be specified for these two programs.

INQ

A **y** in this column indicates that this employee can use Inquire mode for the program or ENTER screen. **n** indicates that the employee cannot.

ADD

A **y** in this column indicates that this employee can use Add mode for the program or ENTER screen. **n** indicates that the employee cannot.

MOD

A **y** in this column indicates that this employee can use Modify mode for the program or ENTER screen. **n** indicates that the employee cannot.

DEL

A **y** in this column indicates that this employee can use Delete mode for the program or ENTER screen. **n** indicates that the employee cannot.

ENTER HELP DOCUMENTATION

Enter Help Documentation lets you create a text file to be associated with a particular program, ENTER screen, or menu. The text file is entered with your favorite TRS-XENIX text editor. The default editor provided with Unify is vi, but you can reset the environment variable, EDIT, to change the editor.

After you select this option (enthdoc) from the MENUH Screen Menu, the screen shows:

```
+-------------------------------------------------------------------+
|[enthdoc]                      UNIFY SYSTEM                        |
|                          5 OCT 1982 - 15:25                       |
|                      Enter Help Documentation                     |
|                                                                   |
|MENU/PROGRAM: _                                                    |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
+-------------------------------------------------------------------+
```

You are asked for either a menu or program name to associate with the file. At the MENU/PROGRAM prompt, enter the name of a

menu, program, or ENTER screen. If the file already exists, it is opened for modification; otherwise, a new file is created. The standard editor commands are used to enter the file. Be sure to press <F1> after you exit the editor.

enthdoc starts the editor with the filename, ../hdoc/xxxxxxxx.n (or $DBPATH/../hdoc/xxxxxxxx.n if DBPATH is set), where xxxxxxxx is the name of the menu, program, or ENTER screen.

The text files you create with Enter Help Documentation are stored in ../hdoc (or in $DBPATH/../hdoc if you have DBPATH set), so this directory must exist before you try to create any help documentation. The system documentation for Unify programs and menus is stored below the Unify library directory, in appl/unify/lib/hdoc (or in $Unify/hdoc if you have Unify set).

When you exit the editor, the MENU/PROGRAM prompt is redisplayed, and you may enter a new filename. <F1> returns you to the MENUH Screen Menu.

PROGRAM LOADING

DISREGARD THIS SECTION UNLESS OPERATING WITH THE TRS-XENIX DEVELOPMENT SYSTEM.

You can control loading of executable modules for an application system by registering each program and executable with MENUH in Executable Maintenance. Note that a program need not appear on a menu to be loaded using this option.

When you select Program Loading (lfilegen) from the MENUH Screen Menu, your screen shows:

```
+----------------------------------------------------------------------+
|[lfilegen]                    UNIFY SYSTEM                            |
|                          5 OCT 1982 - 15:25                          |
|                           Program Loading                            |
|                                                                      |
|PROGRAM NAME: _                                                       |
|                                                                      |
+----------------------------------------------------------------------+
```

At PROGRAM NAME, enter the name of the program you wish to load. This name must match the name of a program in Executable Maintenance. Note that this is not the name of the executable file itself. When the command file is complete, the cursor returns to the prompt, ready to load another executable file.

Program loading is done using one of two modes: automatic or semi-automatic. In automatic mode, you follow the file naming and directory organization conventions so that MENUH can generate the load command file, arrange the order of programs within the executable module, and execute the command file. In semi-automatic mode, you construct your own load command file, leaving MENUH to order the programs and execute the command file.

Automatic and semi-automatic modes have many advantages. MENUH knows all programs and executables in a system, so the Executable Listing Report provides a complete map of the entire applications system, making configuration control easier. It also ensures a consistent directory structure during development of a system, making it easier to maintain programs and train new programmers on a project.

If another person is loading in the same directory, the program waits until the other load is finished before proceeding. This avoids confusion with the a.out file, used by TRS-XENIX as the default output file from loads. Unify keeps you from loading at the same time by calling lock ('l') to create the lock file, LOCKl. If the program is interrupted, this file must be removed before loading again, similar to the TRS-XENIX print spool lock file, /usr/spool/lpd/lock. If you are the TRS-XENIX super user, this scheme does not work, since you can create a file even if it already exists.

The program finds the executable containing the desired program by consulting the data dictionary, mapping the executable name to lowercase letters, and then appending .ld to it. This generates the load command filename. For example, consider the executable module, INVlØØ, containing the programs manlØØ and invlØØ. The load command file, named invlØØ.ld, is executed to load either of the two programs and is then removed.

You can create the command file yourself (semi-automatic mode) or let the program do it for you (automatic mode). If the file is not found, the program generates a command file to accomplish the task. Creating a load command file is done in the following way:

1. The uld command file in the Unify system executable directory is used. $1 is the name of the executable module.

2. If the executable module uses sysrecev as its main entry point, prgtab.o is used as $2. The program automatically generates this file by consulting the executable file layout entered in Executable Maintenance. The file contains a table, menucall, containing pointers to functions, one for each program in the executable file. The source code for the

example executable would appear as:

```
int man1ØØ(), inv1ØØ();
int (*menucall[])() = {
                        man1ØØ,
                        inv1ØØ
                      };
```

The file is compiled automatically to create the .o file, and then both the .c and the .o are removed when the load is complete.

3. For each program in the executable, it is assumed that there is an archive containing the functions for each program. The archives are assumed to be in src subdirectories, named as described earlier in this section. All archives in the src subdirectory are included. The maximum number of archives that Unify can load from one directory is 8, but we recommend you use only one archive per directory.

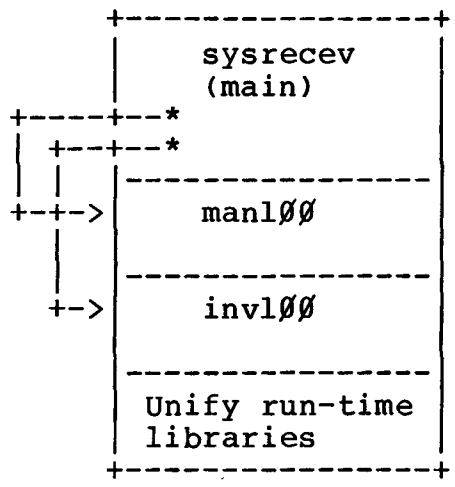
4. Functions used in more than one program may be collected in an archive named util.a so that there is only one copy of the source code. This archive should be placed within a src subdirectory named util. If the src/util subdirectory exists, the archive util.a is included in the command file.

Following these steps, the load command file generated automatically for the example executable file would be called inv1ØØ.ld and would read as:

```
uld     INV1ØØ \
        prgtab.o \
        ../src/man/*.a \
        ../src/inv/*.a \
        ../src/util/util.a
```

The executable module this creates would then contain the programs needed to perform database maintenance for manufacturer and item records. If you do not wish to follow the above steps, custom load files can be created in the appl/unify/bin directory by using a text editor. Since the program looks first to see if a load file exists, your own command file would be used.

UNIFY EXECUTABLE FILE - INV1ØØ



A summary of the steps necessary for program loading follows.

1. Set up the proper directories. You must create the bin, def, and src subdirectories in the directory which contains the database.

2. Build your source directories. For each program or group of programs, make a directory within the src directory, giving them the same name as the first three characters of the program name (may be changed to another name in Executable Maintenance).

3. Build the screens. If the programs use screens, you must build the screens and process them, using the program in the

SFORM Menu, before compiling the programs.

4. Compile your sources and build archives. For each source module, compile the program or subroutine by using:

   ucc -c -Ø program.c

   This produces a module called program.o Then, create an archive in the same directory containing all the object modules with:

   ar {rcv} program.a program.o ...

   Note that if the program uses sysrecev (which it must, if you are loading several programs as one executable) or if you are loading screens, your program name must not be main (). It must be the name you enter in Executable Maintenance.

5. Return to the bin directory and run unify. Go to execmnt (Executable Maintenance) at the MENUH Menu, and add the name of the executable. Enter the program name as you wish it to appear on menus. If you are using sysrecev, enter y. Otherwise, enter n to the SYSRECEV ? prompt. Enter the name of each program within this executable, and show the associated screen if you want it displayed automatically before the execution is turned over to the program. Enter the source directory if it is different from the default (the first three characters of the program name).

6. Load the program. Use lfilegen (Program Loading) at the MENUH Menu. Enter the name of the program to be loaded. If it uses sysrecev, Program Loading builds prgtab.c, compiles it to get prgtab.o, loads it as **main ()**, along with your screens and program, and when finished, sizes the program and moves it to the bin directory. If the program does not use sysrecev, the program is only loaded, sized, and moved to the bin directory.

7. Execute the program. You may now enter the program in your menus or execute it directly by typing its name at a menu selection prompt.

SYSTEM PARAMETER MAINTENANCE

System Parameter Maintenance stores basic parameters for the menu system. Information about the super user, the system title, and mnemonic codes for months of the year can be changed. Since it can be used to change the super user id, we recommend that access to System Parameter Maintenance be restricted with Employee Maintenance. Select System Parameter Maintenance (parmnt) from the MENUH Screen Menu, and the following screen appears:

```
[parmnt]                    UNIFY SYSTEM
                          5 OCT 1982 - 15:25
                      System Parameter Maintenance

SUPER USER ID   : su

ENTRY POINT     : entmenu

SYSTEM HEADING  : UNIFY SYSTEM

LANGUAGE        : EN

MONTH MNEMONICS: JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC

BLOCKS / VOLUME: 849
```

You may change the displayed parameters in any way you wish. Following is an explanation of each prompt on the screen.

SUPER USER ID

"su" indicates the user root. If you use a Unify user ID other than su, the ID **must** be either 3 or 4 characters.

Note: The super user cannot be defined in Employee Maintenance.

ENTRY POINT

The name of the menu seen by the super user after logging in.

SYSTEM HEADING

A 5Ø-character string to be displayed on the first line of all screens and menus in the application system.

LANGUAGE

A two-character code passed by MENUH to programs. sysrecev places the value in a global variable (char langtp[2]), where it is accessible to custom programs.

MONTH MNEMONICS

A 36-character string, containing three characters for each month of the year. This string is used in displaying the second line (with the date and time) on all screens and menus in the application system.

BLOCKS / VOLUME

An integer that tells Write and Read Database Backup and Reconfigure Database the number of 4k (4Ø96 bytes) blocks available on one diskette.

The cursor is at the SUPER USER ID prompt. Pressing <ENTER> moves the cursor down the screen, and <F1> moves it up. <F1> at the first prompt returns control to MENUH. Change an entry by typing the new data and pressing <ENTER>.

Menu Handler Reports

This section describes the standard data dictionary reports furnished with Unify and listed in the MENUH Report Menu. Since the data dictionary is itself a Unify database, it can be queried to produce any information you want. However, these reports satisfy most common needs in listing data dictionary information.

Note that all sample reports in this section have been abbreviated in width for convenience.

EXECUTABLE LISTING

Executable Listing prints a report of all executables in the data dictionary. Also listed are all programs in each executable. Print the listing by selecting it from the MENUH Reports Menu. The report is 93 characters in width. Following is a sample Executable Listing.

DATE : 1Ø/Ø6/82 TIME : 17:11:21 PAGE: 1

SCHEMA REPORTS
Executable Listing

EXECUTABLE	SYSFLAG	PROGRAM	HEADING	IDX	SCREEN	DIRECTORY
ENTER		smodlØØ	Model Maintenance	Ø	smodlØØ	
		smanlØØ	Manufacturer Maintenance	Ø	smanlØØ	
		sitmlØØ	Item Maintenance	Ø	sitmlØØ	
ITM3ØØ	N	itm3ØØ	Item Listing	Ø		
ITM31Ø	N	itm31Ø	Item Aging Report	Ø		
MAN3ØØ	N	man3ØØ	Manufacturer Listing	Ø		
MOD3ØØ	N	mod3ØØ	Model Listing	Ø		
ORDlØØ	Y	ordlØØ	Order Maintenance	Ø	sordlØØ	ord
QUERY	N	query	Query Processor	Ø		

MENU LISTING

Menu Listing prints a list of all menus in the data dictionary. The menus are listed in alphabetical order by name, and each one is followed by the selection options it contains. To print the report, select Menu Listing from the MENUH Report Menu. The report is 79 columns in width. Following is a sample Menu Listing.

DATE : 1Ø/Ø6/82 TIME : 17:11:35 PAGE: 1

SCHEMA REPORTS
Menu Listing

MENU		HEADING / PROMPT	MENU/PROG	M/P
entmenu		Main Menu		
	1	Query Processor	query	P
	2	File Maintenance Menu	fmntmen	M
	3	Report Menu	rptmenu	M
	4	System Menu	sysmenu	M
fmntmen		File Maintenance Menu		
	1	Manufacturer Maintenance	smanlØØ	E
	2	Model Maintenance	smodlØØ	E
	3	Item Maintenance	sitmlØØ	E
rptmenu		Report Menu		
	1	Manufacturer Listing	man3ØØ	P
	2	Model Listing	mod3ØØ	P
	3	Item Listing	itm3ØØ	P
sysmenu		System Menu		
	1	Schema Maintenance	schent	P
	2	Schema Listing	schlst	P
	3	Create Database	crdb	P
	4	SFORM Menu	sfmenu	M
	5	ENTER Screen Registration	entmnt	P
	6	Database Test Driver	sys92Ø	P
	7	Query Processor	query	P
	8	MENUH Screen Menu	scrmen	M
	9	MENUH Report Menu	rptmen	M
	1Ø	Reconfigure Database	scom	P
	11	Write Database Backup	budb	P
	12	Read Database Backup	redb	P

TOTAL MENUS: 4

GROUP LISTING

Group Listing prints a list of all employee groups in the data dictionary. The report is in alphabetical order by group id. Following each group is a list of the programs to which the group has access, along with the access privileges for each program. Print the report by selecting it from the MENUH Reports Menu. The report is 79 columns in width. Following is a sample Group Listing.

```
DATE : 1Ø/Ø6/82  TIME : 17:19:15                              PAGE:    1
                          MENU HANDLER REPORTS
                          Group Access Listing

                                                  ACCESS PRIVILEGES
ID    NAME                      ENTRY PT  MENU/PROG   INQ ADD MOD DEL

DEC   Data Entry Clerks         fmntmen   smanlØØ  E  y   y   y   n
                                          smodlØØ  E  y   y   y   n
                                          sitmlØØ  E  y   y   y   n
```

EMPLOYEE LISTING

Employee Listing prints a list of all employees in the data dictionary. The report is in alphabetical order by employee login id. Following each employee is a list of the programs in which the employee has different privileges than the group. Print the report by selecting it from the MENUH Reports Menu. The report is 132 characters in width. Following is a sample Employee Listing.

DATE : 1Ø/Ø6/82 TIME : 17:19:25 PAGE: 1

MENU HANDLER REPORTS
Employee Access Listing

ID	NAME	GROUP	ENTRY PT	SPECIAL ACCESS PRIVILEGES MENU/PROG	ACC	INQ	ADD	MOD	DEL
jhd	John H. Doe	DEC	fmntmen						
mss	Mary S. Smith	DEC	entmenu	smanlØØ	E	y	y	y	y	y
				smodlØØ	E	y	y	y	y	y
				sitmlØØ	E	y	y	y	y	y
				query	P	y				
				manØØ	P	y				
				modØØ	P	y				
				itmØØ	P	y				
				fmntmen	M	y				
				rptmenu	M	y				

PRINT HELP DOCUMENTATION

Print Help Documentation prints the help documentation associated with a menu or program. After you select Print Help Documentation from the MENUH Reports Menu, you are asked to select a menu or program. Next, the documentation associated with it is printed. Following is a sample help documentation report.

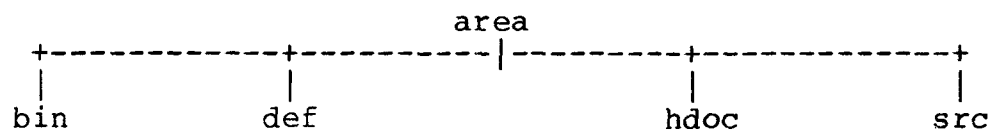
Item Listing

This report lists items in inventory, sorted by description. There is a subtotal for each type of item, and a total count of items at the end. The selection options are a range of dates during which the items were added to inventory.

UNITRIEVE DATABASE MAINTENANCE

This chapter describes how to set up and maintain a UNITRIEVE database. You learn how to maintain the database description, or schema; maintain the database itself (creating, reconfiguring, secondary indexes, and physical layout); establish a secure environment; obtain statistics on database utilization; and use operational utilities.

Create your schema within a directory structure suitable for the program development you want to use once the database is established. The following diagram illustrates a suggested directory structure for applications program development. This is not a required organization, but only an example.



The bin directory contains the data dictionary file, the program executable files, the database file, and other supporting files required at run time. The def directory contains the files "included" in the program source code. These files contain system-wide data definitions and parameters. The src directory contains the source code for the programs. For large systems with multiple modules, the src directory should have module subdirectories.

Schema Maintenance

This section describes the use of the interactive schema entry program, Schema Maintenance. This program can be used either to enter an entirely new schema or to modify the schema of an existing database. The schema is maintained in the Unify data dictionary, in the same UNITRIEVE database file that holds all system menus, screen, programs, employees, and access privileges. Changes you make using Schema Maintenance are not put into effect in the applications database until the database is reconfigured.

Schema Maintenance uses two screens. The first screen is for entering general information about the record types in the database design. The second screen lets you define the fields in each record. When the first screen is complete, the second is displayed.

The field entry screen of Schema Maintenance enforces the rules of schema construction as you enter information. Errors are either reported immediately or not allowed -- the cursor skips over invalid attributes. If you cannot enter a particular attribute, change one of the other attributes of the field to make your entry valid.

Certain changes to the schema are restricted. These are:

1. The key field must always be the first field in the record. The program automatically keeps the key field at the top of the list of fields.

2. A key field cannot be modified or deleted if fields in other records make explicit reference to this field. If you modify a non-referenced key, run Hash Table Maintenance after reconfiguring the database. Otherwise, you cannot access the affected record type by its primary key.

3. Fields can be moved from record to record, but data in a transferred field is not moved.

4. Although a field may be made longer or shorter, its type cannot change. For example, an AMOUNT 4 field can become an

AMOUNT 10 but not a STRING 8.

Select Schema Maintenance from the System Menu to develop your records and fields. The remainder of this chapter describes how to use the screens and list the completed database design.

RECORD MAINTENANCE

The Record Maintenance screen lets you add, modify, and delete record types in the database design. When you select Schema Maintenance (schent) from the System Menu and press <ENTER>, the screen displays:

```
+---------------------------------------------------------------------+
|[schent]                    UNIFY SYSTEM
|                         5 OCT 1982 - 15:25
|                         Schema Maintenance
|
|DATABASE ID: 1
|
|LN  CMD  RECORD   EXPECTED  LONG NAME        DESCRIPTION
|
|     '   '''''''' ''''''    ''''''''''''''''  ''''''''''''''''''''
|
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|""   "   """""""" """"""    """"""""""""""""  """"""""""""""""""""
|
|[N]ext page, [P]rev page, [A]dd line, or number _
|
+---------------------------------------------------------------------+
```

'''''''' = record entry area
"""""""" = record paging area

New entries are added at the top of the screen. Existing entries are displayed and modified in a page format, in the area below. Each "page" displays 11 entries.

Following is an explanation of each column heading and prompt.

[N]ext page, [P]rev page, [A]dd line, or number

Lets you choose a mode for the paging area. The modes are:

n -- Displays the next page of database record types.

p -- Displays the previous page of database record types.

a -- Lets you add new database record types by moving the cursor to the record entry area. Press <F2> to store a new record type.

number (1-99) -- Displays the page containing the indicated database record type and positions the cursor at that record to let you modify or delete it.

LN

A line number assigned by the system, used to reference the line if you wish to modify or move it.

CMD

A column used for entering a command to perform an operation on the current line. The cursor moves up and

down in the paging area in response to <F1> and <ENTER>. Valid commands are:

f -- Modify the fields for the current record type by going to the second screen.

m -- Modify EXPECTED, LONG NAME, or DESCRIPTION for the current record.

d -- Delete the current record.

number (1-99) -- Renumber the current record as indicated, and reorder the records on the screen.

q -- Redisplay the paging prompt at the bottom of the screen.

RECORD

A unique eight-character record name. It can contain only letters (upper- and lowercase), numbers, and underscores (_), and it must begin with a letter. The record name references this record throughout the system design.

EXPECTED

The expected number of records of this type.

LONG NAME

A more descriptive record name of up to 16 characters, used by ENTER and the database test driver in displaying error messages. It can contain only letters (upper- and lowercase), numbers, and underscores (_), and it must begin with a letter.

DESCRIPTION

A comment field used for entering a description of the record.

To add a new record, enter a at the bottom of the screen to select Add mode. The prompt is erased, and the cursor moves to the entry area directly under the RECORD column. <ENTER> moves the cursor forward, and <F1> moves it backward. <F2> commits the record and redisplays the prompt.

FIELD MAINTENANCE

This section explains the use of the Field Maintenance screen. Any record's fields may be added, modified, and deleted. When the Record Maintenance screen is completed, the following screen appears:

```
+-----------------------------------------------------------------------+
|[schent]                   UNIFY SYSTEM                                 |
|                        5 OCT 1982 - 15:25                              |
|                        Schema Maintenance                              |
|                                                                        |
|RECORD: xxxxxxxx                                                        |
|                                                                        |
|LN  CMD  FIELD     KEY  REF      TYPE    LEN LONG NAME    COMB. FIELD  |
|                                                                        |
|     '  ''''''''    '  '''''''' '''''''  ''' ''''''''''''   ''''''''    |
|                                                                        |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|""    "  """"""""    "  """""""" """""""  """ """"""""""""   """"""""    |
|                                                                        |
|[N]ext page, [P]rev page, [A]dd line, or number:_                       |
|                                                                        |
+-----------------------------------------------------------------------+
```

'''''''' = field entry area
"""""""" = field paging area

This screen works in a similar way to the first one, with an entry area and a paging area. The prompts are:

[N]ext page, [P]rev page, [A]dd line, or number:

Lets you choose the mode for the paging area. The modes are:

n -- Displays the next page of fields for the current record type.

p -- Displays the previous page of fields for the current record type.

a -- Lets you add new fields by moving the cursor to the field entry area. Press <F2> to store a new field.

number (1-99) -- Displays the page with the indicated field, and positions the cursor at that field to let you modify or delete it.

RECORD

A display only prompt that shows you the record in which you are adding or modifying fields.

LN

A line number assigned by the system, used to reference the line if you wish to modify or move it.

CMD

A column used for entering a command to perform an operation on the current line. The cursor moves up and down in the paging area in response to <F1> and <ENTER>. Valid commands are:

m -- Modify KEY, REF, TYPE, LEN, LONG NAME, or COMB. FIELD for the current field.

d -- Delete the current field.

number (1-99) -- Renumber the current field as indicated, and reorder the fields on the screen.

q -- Redisplay the prompt at the bottom of the screen.

FIELD

A unique eight-character field name. It can contain only letters (upper- and lowercase), numbers, and underscores (_), and it must begin with a letter. The field name references the field throughout the system design.

KEY

An asterisk (*) in this column indicates that this field is the primary key of the record. The primary key is used to produce an index on the record to facilitate searches when a query is performed and to check for uniqueness when data is added.

The key field must always be the first field in the record. The program automatically keeps the key field at the top of the list of fields.

A key field cannot be modified or deleted if fields in other records make explicit reference to this field. If you modify a non-referenced key, run Hash Table Maintenance after reconfiguring the database. Otherwise, you cannot access the affected record type by its primary key.

Secondary indexes may also be defined on a record to speed up the search process. See Secondary Index Maintenance in this chapter for more information.

REF

The name of the primary key of another record. When the name of a primary key is placed in this column, an explicit relationship with another record type is created. For more information about explicit relationships, refer to the Unify Tutorial Manual, Chapter 2, "Entering a database design."

TYPE

The data type of the field. Valid data types are:

N -- NUMERIC
F -- FLOAT
S -- STRING
D -- DATE
T -- TIME
A -- AMOUNT
C -- COMB (combined field)

A combined field defines a name to be associated with a group of fields. For example, individual fields may be combined within the primary key.

LEN

The default display length of the field on screens and reports. For NUMERIC and STRING fields, LEN is exactly the display length. For FLOAT fields, LEN has the form nnd. nn indicates the total number of display positions, including the decimal point, and d indicates the number of digits to the right of the decimal point. For example, a FLOAT field with a LEN of 123 has a default format of:

nnnnnnnn.nnn

For AMOUNT fields, LEN is the number of digits to the left of the decimal point. The two digits to the right of the decimal point are assumed. For example, an AMOUNT field with a LEN of 6 appears on screens and reports as:

nnnnnn.nn

Maximum lengths for field types are:

```
NUMERIC -- 9
FLOAT   -- 2ØØ
STRING  -- 256
DATE    -- defaults to 8
TIME    -- defaults to 5
AMOUNT  -- 11
COMB    -- computed by system
```

LONG NAME

A more descriptive name of up to 16 characters for the field. It can contain only letters (upper- and lowercase), numbers, and underscores (_), and it must begin with a letter. SQL uses this name to identify fields when performing queries. The name does not have to be unique throughout the database but must be unique within this record type.

COMB. FIELD

The name of a combined field in the current record of which this field is a part.

To add new fields to the current record, enter a at the bottom of the screen to select Add mode. The prompt is erased, and the cursor moves to the entry area directly under the FIELD column. Similar to the first screen, <ENTER> moves the cursor forward, and <F1> moves it backward. <F2> commits the field and redisplays the prompt.

SCHEMA LISTING

Selecting the Schema Listing option gives you a list of the current schema. All records, fields, and relationships are printed as well as alphabetical cross references of records and fields. The report is 79 characters in width. A sample Schema Listing follows.

DATE : 10/15/82 TIME : 11:29:35 PAGE: 01

SCHEMA REPORTS
Schema Listing

RECORD/FIELD		REF	TYPE	LEN	LONG NAME
manf	10				manufacturer
	*mano		NUMERIC	4	number
	mname		STRING	35	name
	madd		STRING	30	address
	mcity		STRING	20	city
	mstate		STRING	2	state
	mzip		NUMERIC	5	zip_code
model	50				model
	*mokey		COMB		model_key
	monum		NUMERIC	7	number
	momano	mano	NUMERIC	4	manufacturer
	mdes		STRING	30	descr
item	100				inventory_item
	*sno		NUMERIC	9	number
	imodel	mokey	COMB		model_key
	iad		DATE		acquisition_date
	isal		AMOUNT	5	sales_price
	iorder	onum	NUMERIC	9	order_number
	ipamt		AMOUNT	5	purchase_price
customer	10				customer
	*cnum		NUMERIC	5	customer_number
	cname		STRING	30	name
	caddr		STRING	30	address
	ccity		STRING	20	city
	cstate		STRING	2	state
	czip		NUMERIC	5	zip_code
	cphone		STRING	14	phone_number
orders	100				order
	*onum		NUMERIC	9	order_number
	odate		DATE		date_ordered
	ocust	cnum	NUMERIC	5	customer_number

The number to the left of the REF column in the above table indicates the expected number of records for that table.

DATE : 10/15/82 TIME : 11:29:46 PAGE: 02

SCHEMA REPORTS
Record List

RECORD	NUMBER	EXPECTED	LENGTH
customer	4	10	55
item	3	100	15
manf	1	10	50
model	2	50	21
orders	5	100	8

DATE : 10/15/82 TIME : 11:29:49 PAGE: 03

SCHEMA REPORTS
Field List

FIELD	NUMBER	RECORD	TYPE	LENGTH
caddr	21	customer	STRING	30
ccity	22	customer	STRING	20
cname	20	customer	STRING	30
cnum	19	customer	NUMERIC	5
cphone	25	customer	STRING	14
cstate	23	customer	STRING	2
czip	24	customer	NUMERIC	5
iad	12	item	DATE	
imodel	9	item	COMB	6
imodel_momano	11	item	NUMERIC	4
imodel_monum	10	item	NUMERIC	7
iorder	17	item	NUMERIC	9
ipamt	18	item	AMOUNT	5
isal	13	item	AMOUNT	5
madd	3	manf	STRING	30
mano	1	manf	NUMERIC	4
mcity	14	manf	STRING	20
mdes	7	model	STRING	30
mname	2	manf	STRING	35
mokey	4	model	COMB	6
momano	6	model	NUMERIC	4
monum	5	model	NUMERIC	7
mstate	15	manf	STRING	2
mzip	16	manf	NUMERIC	5
ocust	28	orders	NUMERIC	5
odate	27	orders	DATE	
onum	26	orders	NUMERIC	9
sno	8	item	NUMERIC	9

NUMBER refers to the order in which the fields appear in the Schema Listing.

DATE : 10/15/82 TIME : 11:29:53 PAGE: 04

SCHEMA REPORTS
Record Relationships

RECORD	RECORD	FIELD
item	model	imodel
item	manf	imodel_momano
item	orders	iorder
model	manf	momano
orders	customer	ocust

The record relationships defined in this report can be interpreted as RECORD references RECORD through FIELD. For example, the item record references the model record through the item field imodel.

The item record references the manf record by way of the model record implied in one part of the combined field imodel_momano.

DATE : 10/15/82 TIME : 11:30:00 PAGE: 05

SCHEMA REPORTS
Field Change List

FIELD	CHANGE
ocust	A
odate	A
onum	A
opoo	D

This report only appears if you run Schema Listing after making changes to the screen and have not yet reconfigured the database. Only the fields that have been added or deleted since the last reconfigure are reflected in this report.

Database Maintenance

Creating a database, database reconfiguration, secondary index maintenance, and volume maintenance are all part of creating and maintaining the physical structure of a database. These utilities are used in UNITRIEVE database management, and all are discussed in this section.

CREATE DATABASE

The Create Database utility creates an initial database file from a schema description in the data dictionary. The creation process consists of compiling the schema, determining the type of file in which the database is to reside, and writing the initial file. These steps are all performed automatically by the program. Since Create Database creates an empty database file, you should observe the following precautions:

1. No one else should use your Unify application when Create Database is running. Make sure that no one else is using the application before you start.

2. The database should not contain any valuable data, since it will be empty when Create Database finishes. If you have valuable data, you may backup the database by executing Write Data Base Backup and then reload the data with Read Data Base Backup when the database is created, or you may dump all the data using SQL then reload the data after the database has been created by executing Database Load.

Errors are possible during the schema compiling process. To guard against losing your schema, the data dictionary is copied to a backup file before the process begins. If a fatal error occurs, it is restored from this backup copy.

A UNITRIEVE database file can reside in either an ordinary or special TRS-XENIX file. The type of file is determined by setting up the appropriate parameters using Volume Maintenance. The default is an ordinary TRS-XENIX file in the current

directory. The filename assigned by the create program is file.db. If an ordinary TRS-XENIX file is used, file.dbr is linked to it. Otherwise, file.dbr is set to point at the raw device on which the data file resides. Thus, file.dbr has meaning only if you are using a special file.

Be sure that file.db and file.dbr always point to the same TRS-XENIX file. Ordinarily, this is not a problem. However, some TRS-XENIX utilities, such as **cp** from one directory to another, create two separate files from files that are linked together. You can see if this has happened by doing an **ls -i file.db fil.dbr** to see if both files have the same inode number. If they do not, remove file.dbr, and link them again, using the command, **ln file.db file.dbr**.

If you expect the size of your data file to exceed one megabyte, consider creating the database as a special TRS-XENIX file. This gives you greater speed and lets you process more information by letting programs perform I/O on the raw device, bypassing the TRS-XENIX file indexing and buffering schemes. Refer to "Volume Maintenance" to learn how this is done.

To create an empty data file, select Create Database (crdb) from the System Menu. The screen shows:

```
+-------------------------------------------------------------------------+
|[crdb]                          UNIFY SYSTEM                             |
|                          5 OCT 1982 - 15:25                             |
|                           Create Database                               |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
|PROCEED? _                                                               |
+-------------------------------------------------------------------------+
```

Press **y** to start creating the database. (n or <F1> stops creation and returns to the menu handler.) The program displays the prompts shown on the next screen as the creation proceeds.

```
[crdb]                          UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                              Create Database


** Phase I   **

** Phase II  **

** Phase III **









Process complete.  Back up the database ->->_
```

When the process is complete, press <ENTER> to answer the final prompt, and the System Menu is redisplayed.

RECONFIGURE DATABASE

A UNITRIEVE database can be reconfigured to add or delete record types, fields, and relationships between record types. You can also lengthen or shorten fields or increase and decrease the expected number of records. See "Schema Maintenance" for the restrictions that apply. The data dictionary contains the information necessary for reconfiguring the database.

Reconfigure Database reformats the database file and updates the data dictionary. While reformatting is taking place, no one

else should be allowed to use your Unify application. Make sure that no one else is using your application before you start.

The reconfiguration program goes through a number of steps to perform its job. First, it locks out all database updates. Next, it makes a copy of the data dictionary so that if a fatal error occurs (such as a read/write error), the dictionary can be reset to its original state. Next, it compiles the schema and generates tables to translate records from the old format to the new. The database is written to a temporary file and then read back in its new format.

If you increase or decrease the expected number of records, you can rebuild the hash table index. If the nominal hash table loading factor is greater than 70%, you should rebuild the hash table index. If you change the internal data type of the primary key field using Schema Maintenance, you must rebuild the hash table index by running Hash Table Maintenance after you reconfigure the database. Otherwise, you cannot access the affected record type by its primary key.

To reconfigure an existing database, first make sure that you have a current backup of the database. See "Write Database Backup" for instructions. Then, select Reconfigure Database (scom) from the System Menu. The screen shows:

```
+-------------------------------------------------------------------+
|[scom]                          UNIFY SYSTEM                       |
|                          5 OCT 1982 - 15:25                       |
|                          Reconfigure Database                     |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|PROCEED? _                                                         |
+-------------------------------------------------------------------+
```

Press y at the PROCEED? prompt to start the process. n or <F1> stops the process and returns you to the System Menu. After you press y, the screen shows:

```
[scom]                          UNIFY SYSTEM
                            5 OCT 1982 - 15:25
                          Reconfigure Database


** Phase I   **

** Phase II  **



Nominal hash table loading factor:  5Ø%
New load factor without rebuilding: nn%









REBUILD HASH INDEX? _
```

At this point, the schema has been compiled, and the actual reconfiguration is about to begin. If you have increased the total expected number of records in the database, you have an opportunity to increase the size of the hash table here. Press either **y** or **n** at the REBUILD HASH INDEX? prompt. Next, you see the following prompts:

```
[scom]                          UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                            Reconfigure Database


** Phase I   **

** Phase II  **

** Phase III **


Nominal has table loading factor:  5Ø%
New load factor without rebuilding: nn%









USE DISK AS THE TEMPORARY FILE? _
```

This question gives you a chance to store the reconfiguration on diskette. Press either y or n to answer this prompt.

When the process is complete, back up the database again by selecting Write Database Backup (budb). The database file is now reconfigured, and the data dictionary is updated to reflect the new schema design.

You need not recompile or reload existing programs, since the schema description is bound dynamically. However, you should re-process screen forms, because they contain information about the database fields.

If you change the internal data type of the primary key field using Schema Maintenance, you must rebuild the hash table index by running Hash Table Maintenance after you reconfigure the database. Otherwise, you cannot access the affected record type by its primary key.

SECONDARY INDEX MAINTENANCE

In addition to primary keys, secondary keys (indexes) may be defined on a field of a record to speed up the search process when a query is performed. Before you use Index Maintenance, make sure that no one else is updating the field(s) for which you want to add or drop indexes. To add a secondary (B-tree) index or to drop an existing index, go to the Database Maintenance Menu, and select Index Maintenance (idxmnt). The screen shows:

```
[idxmnt]                        UNIFY SYSTEM
                              1 OCT 1982 - 13:ØØ
                              Index Maintenance

[A]DD OR [D]ROP INDEX :
FIELD NAME            :




[A]SCENDING OR [D]ESCENDING:
```

Following is an explanation of each prompt:

[A]DD OR [D]ROP INDEX:

The cursor is at this prompt when the screen appears. The two selections are:

a -- Add a B-tree index for a database field.

d -- Drop a database field's B-tree index.

FIELD NAME:

Enter the name of the database field in which a B-tree

index is being added or dropped. The field cannot be a COMB field. After you enter the field name, a confirmation message appears.

[A]SCENDING OR [D]ESCENDING:

This question appears only if you are adding an index.

a -- Specifies that entries in the index should be in ascending order.

d -- Specifies that entries in the index should be in descending order.

At the [A]DD OR [D]ROP INDEX prompt, <F1> returns you to the menu handler. Otherwise, <F1> positions the cursor at the previous prompt.

When a response has been provided for each of the prompts, press <F2> to record the index information.

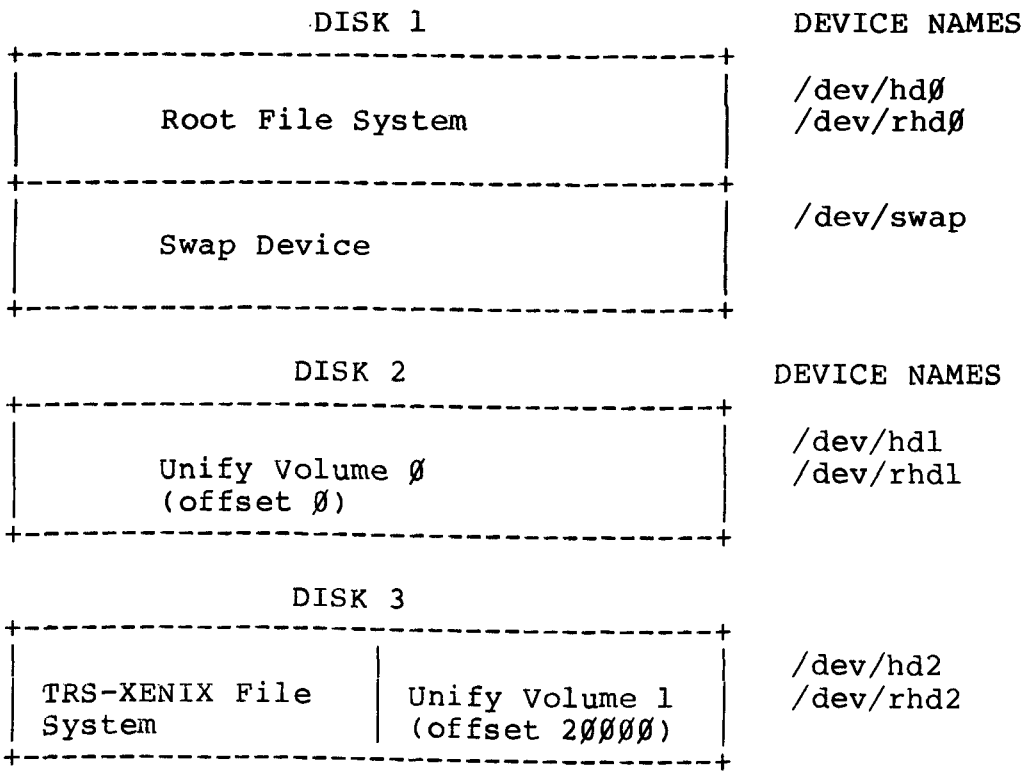
While a B-tree index is being constructed, a counter is displayed, indicating the number of field occurences (in groups of 10) that have been processed.

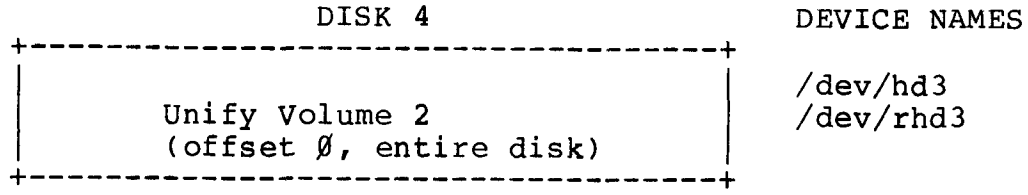
VOLUME MAINTENANCE

You can store a database in one or more contiguous extents of diskette space ("raw disk"). These extents are called "file systems" in TRS-XENIX terminology. Normally, file systems are initialized using mkfs and mounted as directories in the TRS-XENIX file hierarchy. Instead of using them as TRS-XENIX file systems, you can use contiguous extents of disk space as part of a Unify database. This adds to the efficiency of systems designed using Unify. If no disk extents are specified, the database is created as an ordinary TRS-XENIX file in the current directory.

Use Volume Maintenance to describe the extents you want to use for the database. These extents are called "volumes." The Create Database program looks at specified volumes and initializes them for use. Volume Ø always contains the database parameters and the hash table. It must always have an offset of Ø from the start of the specified file system. Other volumes may have offsets other than Ø.

The following layout shows how a four-disk TRS-XENIX system might be configured using Unify.





Device hdØ contains the root file system. Device swap contains the swap area. The second disk holds the root volume of the Unify database. Unify uses both the block device (/dev/hdl) and character device entries (/dev/rhdl). The first part of the third disk contains a TRS-XENIX file system, created by running mkfs with the appropriate number of blocks. The remainder of the device (/dev/hd2) holds the second volume of the Unify database. The entire fourth disk (/dev/hd3) contains the remainder of the Unify database.

Any number of Unify databases can be on your computer system. Some of them may be on raw devices, with the rest in ordinary TRS-XENIX files. It is up to you to prevent collisions in device assignments.

If you do not specify devices, Create Database uses an ordinary file. Links to the file are created with the names file.db and file.dbr. If you specify special devices in Volume Maintenance, Create Database (and Reconfigure Database) create file.db and file.dbr as special files. file.db is the block-style device for the root device, and file.dbr is the character-style device for the root device.

To specify database volumes, select Volume Maintenance (volmnt) from the Database Maintenance Menu. The screen shows:

```
[volmnt]                        UNIFY SYSTEM
                              5 OCT 1982 - 15:45
                              Volume Maintenance

                    CHARACTER      BLOCK
 CMD  NAME  RT      DEVICE         DEVICE          OFFSET      LENGTH

  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
  '    ''''  '  '''''''''''''  '''''''''''''  '''''''''  '''''''''
```

Each line on the screen represents one volume of the database. When you next display current volumes, blank lines will not appear. One line must contain an X in the RT column, indicating that this is the root device. The root device must have an offset of Ø blocks.

To add a new volume, move the cursor to an empty line, and type a <ENTER> in the CMD column. To modify an existing volume, move the cursor to the CMD column for that volume, and type m <ENTER>. To delete an existing volume, move the cursor to the CMD column for that volume, and type d <ENTER>.

In the CMD column, <ENTER> alone moves the cursor down the page, and <F1> moves it up the page. <ENTER> at the last line

moves the cursor to the first line. <F1> at the first line exits Volume Maintenance.

When you add or modify a line, <ENTER> moves the cursor to the right, and <F1> moves the cursor to the left. <ENTER> at the last column moves the cursor to the first modifiable column on the line. <F2> at the first modifiable column stores the entry and moves the cursor to the CMD column on the next line.

Following is an explanation of each column.

CMD

When the cursor is in this column, you may add a new volume or modify/delete the volume on the current line.

NAME

A four-character name for the volume. It is used primarily for display and error reporting purposes. The name must be unique.

RT

Typing **x** in this column signifies that this is the root volume. The root volume contains the hash table and the database parameters.

CHARACTER DEVICE

The character device name for the volume -- for example, /dev/rhd1.

BLOCK DEVICE

The block device name for the volume -- for example, /dev/hd1.

OFFSET

The offset (the number of 512-byte blocks) from the start of the device.

LENGTH

The length (the number of 512-byte blocks) of the volume.

Security

This section explains how to control access to a Unify database in a secure environment. Unify provides several levels of security, which, when used in conjunction with the TRS-XENIX protections, can be used to create a very secure environment.

At the highest level, Unify provides program and menu secuity so that unauthorized users cannot run restricted programs or menus. Next, field level security prevents unauthorized access to individual fields within the Listing Processor, SQL, and the Host Language Interface. Finally, the TRS-XENIX file protection scheme prevents users from dumping the database contents to the screen or printer.

FIELD-LEVEL SECURITY

Specify field-level security by using the Field Security Maintenance screen. This screen lets you define separate read and write passwords for each database field. Once specified, the field can only be read or written if the appropriate password is supplied.

Once you define or change passwords, you must process them to produce a binary file containing the password information. From then on, the user must supply the password to access the field. (See the explanations of **unlock** and **accsfld** in the section of this Reference Manual on C Language Interface.) If a read-only password is specified, users are given only read privileges when they supply this password. If a write-only password is specified, users are given only write privileges when they supply the password. To set a password on a combined field, the password must be assigned to each component.

In addition to passwords, fields are given a security group in both the read and write categories. If a user supplies the password for any member of the group, the entire group may be read or written, depending upon the password. If a group is not specified, the field is assumed to be in a group by itself.

A user program must have write privileges on all fields in a file to add or delete records in that file.

To use the program, select Field Security Maintenance (fldsec) from the Database Maintenance Menu. The screen shows:

```
[fldsec]                        UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                          Field Security Maintenance

 FIELD NAME:

 READ ONLY PASSWORD:
 READ ONLY GROUP    :

 WRITE PASSWORD     :
 WRITE GROUP        :









 [A]DD, [I]NQUIRE, [M]ODIFY, [D]ELETE _
```

Following is an explanation of each prompt.

[A]DD, [I]NQUIRE, [M]ODIFY, [D]ELETE

Specifies the mode of the program. Enter **a** for Add mode, **i** for Inquire mode, **m** for Modify mode, or **d** for Delete mode. This prompt may vary, depending upon your access privileges.

FIELD NAME:

The name of the field to have its password attributes modified.

READ ONLY PASSWORD:

The password giving a user read access on this field. If no write password is specified, this password gives the user both read and write privileges for the field.

READ ONLY GROUP:

The name given to a group of fields for security purposes. All fields for which the same read only group name has been specified are in the same group. Providing the read only password for any field in the group permits the entire group of fields to be read and/or written.

WRITE PASSWORD:

If this password is specified, the user must supply it to have write privilege on the field. If the write password only is specified, the field can be both read and written.

WRITE GROUP:

The name given to a group of fields for security purposes. All fields for which the same write group name has been specified are in the same group. Providing the write password for any member of the group permits the entire group of fields to be read and written.

PROCESS FIELD PASSWORDS

The Process Field Passwords program processes the field-level password specifications to produce the binary password file, upasswd.

To process the current set of field-level passwords, select Process Field Passwords (procpass) from the Database Maintenance Menu. The screen shows:

```
[procpass]                    UNIFY SYSTEM
                          5 OCT 1982 - 15:25
                        Process Field Passwords









PROCEED? _
```

Pressing y produces the upasswd file in the current directory (or in $DBPATH if this environment variable is set).

n or <F1> returns you immediately to the menu handler without any processing.

A SECURE UNIFY ENVIRONMENT

The measures described in the previous sections are only as secure as the TRS-XENIX environment in which they operate. If users are free to run programs designed to defeat security, or if they can dump the contents of database files to a screen or printer, they can access unauthorized data. This section explains how to create a secure environment using Unify.

The first task is assigning privileges with the menu handler. The Unify super user password must be kept secure and used only by the highest-level user. See "System Parameter Maintenance" for instructions on modifying the super user's id and password. We recommend that various database functions such as Schema Maintenance, Write Database Backup, Read Database Backup, and Reconfigure Database be assigned to other ids. This lessens the need for general distribution of the super user id and password. Obviously, you should use care in assigning access to Employee Maintenance and Group Maintenance.

Security in Executable Maintenance is a critical area. Since security is established so that database application programs can run only via Unify, it is important to verify the procedures being used in these programs before they are registered. For example, employees might have read privileges on a check file, but this does not mean that they should be able to register a check printing program.

To prevent access to the database by unauthorized programs, you should establish a special TRS-XENIX user id. Next, all system files for an application, such as unify.db, file.db, file.dbr, and upasswd should be owned by this user id. The owner should have read and write privileges for these files. If the database is to be accessed by multiple users, read and write privileges must be set for the group as well. Next, you should move the unify program to its own directory (along with all other Unify executables), which must be owned by the special user.

Statistics

This section explains how to review database statistics about the size and characteristics of various database parameters. Basic statistics reported include file sizes, per cent of space utilized, hash table density, and collision information.

DATABASE STATISTICS

This program prints statistics about various database parameters. Start it by selecting Database Statistics from the Database Maintenance Menu. The report, which is printed, is 79 characters in width. Following is a sample Database Statistics Report.

DATE : Ø6/17/83 TIME : 11:13:Ø7 PAGE: 1

SCHEMA REPORTS
DATA BASE STATISTICS

RECORD	EXPECTED	ACTUAL	PERCENT
manf	12	6	5Ø.Ø
model	54	15	27.8
item	1Ø2	14	13.7
customer	12	3	25.Ø
orders	1Ø2	4	3.9
node	12	1	8.3

HASH TABLE 7.3% FULL
DATA BASE SIZE 39 BLOCKS

Following is an explanation of each heading.

RECORD

The name of the record type.

EXPECTED

The approximate number of expected records. Due to the space allocation algorithm, this number might not be the exact expected number entered in Schema Maintenance.

ACTUAL

The actual number of records in the file.

PERCENT

The per cent of space filled in the file. A given file can be 166% full before more space must be allocated with a reconfiguration.

HASH TABLE

The per cent of slots filled in the hash table. 50% is the most efficient loading factor. If more than 70% of the slots are filled, performance is less efficient, so execute Hash Table Maintenance to rebuild the hash table index.

DATA BASE SIZE

The size of the database in 512-byte blocks.

HASH TABLE STATISTICS

This program prints statistics about various hash table parameters. Start it by selecting Hash Table Statistics from the Database Maintenance Menu. This report is sent only to the screen. Press <ENTER> when you finish reviewing the report. Following is a sample report.

```
[hts]                          UNIFY SYSTEM
                            5 OCT 1982 - 15:25
                           Hash Table Statistics


                  HASH TABLE STATISTICS

Total Entries              1Ø76Ø1
Per Cent Filled             42.ØØ
Average Chain Length         3.Ø8
Longest Chain Length           51
Starting Address            56896

                      Number of Chains of Length...

2       3       4       5       6       7       8       9       1Ø      >1Ø
3568    14165   18      Ø       Ø       97      466     71      1Ø3     1613




->->_
```

The screen labels have the following meanings:

Total Entries

The total number of slots filled in the hash table.

Per Cent Filled

The per cent of entries filled. When 50% of the slots are filled, the table is at maximum optimal density. If 70% or more of the slots are filled, performance is less efficient, so execute Hash Table Maintenance to rebuild the hash table index.

Average Chain Length

The average number of consecutive slots filled before an empty slot occurs. Half of this number is a gross measure of the number of probes necessary to find a record. In the above sample, it takes an average of 1.5 probes to find a record. The actual number is probably less, since the record type is stored in the hash table and consecutive slots are very likely to be filled with entries for different record types. If this number is greater than six, you should rebuild the hash table index.

Longest Chain Length

The longest sequence of consecutive filled slots. This gives a gross measure of the "worst case" access time.

Starting Address

The address of the first slot of the longest chain, in case you want to review the hash table.

Number of Chains of Length...

The number of sequences of consecutive filled slots of the indicated length. This gives you an idea of the distribution of chain lengths.

Operational Utilities

This section describes Unify's operational utilities. Four different programs help you operate your application to ensure that the database can always be recovered. There is also a program for loading a batch of data into a database. Good procedure requires that you back up your database daily, especially if you update it frequently. Then, if a hardware failure destroys file system integrity, or if a bad block occurs on the diskette, you can restore the database file to a recent state. Write Database Backup and Read Database Backup provide for daily backup and recovery of the schema and all data in the database. The utility usave will save and restore the schema, data, and all screens, reports, indexes, help documentation, and application programs defined within the database directory structure.

If you experience a software failure or a recoverable hardware failure, and you suspect that database integrity has been compromised, utilities are available to rebuild critical parts of the file. The hash table can be rebuilt by using Hash Table Maintenance. If you find that you cannot access a record or group of records by key, run Hash Table Maintenance to correct the problem.

If queries or programs using explicit relationships give you inconsistent results, you can rebuild these by using Explicit Relationship Maintenance. If all these methods fail, you can dump all the data using SQL, use Create Database, and then reload the data using Database Load.

WRITE DATABASE BACKUP

This function makes a backup of the UNITRIEVE database file and the data dictionary in the current directory only. (All files pertaining to screens, reports, indexes, help documentation, and application programs are not saved by this utility.) The program finds the database file and notes whether it is an ordinary TRS-XENIX file or is spread across several raw devices. It then

calculates the number of blocks to write by reading the database size from the file itself and handles multiple diskette backups for large database files.

The default backup device is set by the parameter BUDEV in the /usr/bin/unify script to /dev/rfdØ. You may change the backup device by using a text editor to modify the unify script or by setting this parameter for a specific user in the user's .profile.

Before starting the backup, make sure that no one is using the database in the current directory. This ensures that a good physical copy of the database is made.

To start the program, select Write Database Backup (budb) from the System Menu. The screen shows:

```
+-------------------------------------------------------------------+
|[budb]                        UNIFY SYSTEM                         |
|                            5 OCT 1982 - 15:25                     |
|                         Write Data Base Backup                    |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|PROCEED? _                                                         |
+-------------------------------------------------------------------+
```

Insert a formatted diskette, and make sure that it is write-enabled before responding to the PROCEED? prompt. (Refer to diskutil in your TRS-XENIX Manual.) Press y to start the backup process. (n or <F1> bypasses the process and returns you to the menu.) After the backup is written, press <ENTER> in response to the ->-> prompt to return to the menu. Following are other messages you may see, along with the appropriate action to take.

PROMPT	RESPONSE	PROGRAM ACTION
Database size: xxx 16k blocks	None.	This is the number of blocks to be written. For information only.

PROMPT	RESPONSE	PROGRAM ACTION
Change diskettes ->->	Insert another diskette, and press any key.	Next diskette begins to write.
Diskette is not ready or is read only ->->	Insert diskette or remove write protection, and press <ENTER>.	After correcting the error, diskette begins to write.
xxx 16k records written to disk ->->	Any response.	Backup is complete. Menu is redisplayed.

READ DATABASE BACKUP

This program reads backup diskettes written by the Write Database Backup program to restore UNITRIEVE database and data dictionary files. It assumes the same drive name specified by the BUDEV parameter in the unify script as used by Write Database Backup. Make sure that no one is using the database before you start the recovery process.

To start the program, select Read Database Backup (redb) from the System Menu. The screen shows:

```
+-------------------------------------------------------------------+
|[redb]                          UNIFY SYSTEM                       |
|                            5 OCT 1982 - 15:25                     |
|                          Read Data Base Backup                    |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|PROCEED? _                                                         |
+-------------------------------------------------------------------+
```

Insert the diskette containing the backup. Press **y** at the PROCEED? prompt to start the recovery process. (**n** or <F1> bypasses the process and returns you to the menu.) The backup diskette is read in, and the ->-> prompt is displayed. Press <ENTER> to return to the menu. Following are some additional messages you may see, along with the appropriate action to take.

PROMPT	RESPONSE	PROGRAM ACTION
Database size: xxx 16k blocks	none	Number of blocks to be read. For information only.
Change diskettes ->->	Insert next backup diskette, then press any key.	Next diskette begins to read.
Diskette is not ready or is read only ->->	Insert diskette, and press <ENTER>.	After correcting the error, diskette begins to read.
xxx 16k records read from disk ->->	Any response.	Recovery is complete. Menu is redisplayed.
Clearing remainder of file.dbr	none	Displayed only if the database is on a raw file. Zeroes are being written to the end of the file system. When finished, the menu is redisplayed.

USAVE

The usave utility saves and restores all information in the directory from which Unify was executed, if the directory structure includes the subdirectories bin, hdoc, src, and def. It is not necessary to use this utility on a daily basis. Write and Read Database Backup will handle updates of daily information, but if you make a change to a screen, report, index, help documentation, or application program, you must use usave to backup the information.

You must execute usave from the shell, so if you are in the Unify system, exit to the shell by typing sh. At the new prompt, type:

usave <ENTER>

The following options appear on the screen:

```
Save/Restore Unify Data Files
-----------------------------

1  to save the entire database
2  to restore the entire database
q  to quit
```

Please Select:

To Save

Before selecting the save option, make sure you have a formatted diskette. Select the save option by typing 1 at the select prompt. The following message appears:

```
Enter 1 for single-sided disks 2 for double-sided:
```

Depending on the type of formatted disk you are using, enter either 1 or 2 for the above prompt. A new prompt appears:

```
Enter Drive Number:
```

Enter the number of the disk drive into which you are going to insert the diskette. The following messages appear:

```
Database Save
Insert first disk in drive fd? Then press <ENTER>
```

Insert your formatted diskette into the drive you indicated above, and press <ENTER>.

The screen lists the files being saved from the bin, src, hdoc, and def directories. If usave does not find the directories listed above, the following error message appears:

> Database must have directory structure shown in Unify Reference manual, please run ucreate before executing usave.

After the files are saved, the first screen returns, and you can press **q** to exit the utility.

To Restore

To restore the database, enter **2** at the selection prompt. The following prompt appears:

```
Enter 1 for single-sided disks 2 for double-sided:
```

Enter the appropriate number for the diskettes you are using. The following prompt appears:

```
Enter Drive Number:
```

Indicate the number of the disk drive into which you are going to insert the diskette. The following messages appear:

```
Restore database backup
First diskette ready? (y,n)
```

If you enter **n**, the program returns to the first screen. Type **y** to continue the restore procedure. The screen then shows a list of the files being restored from the bin, hdoc, def, and src directories.

After all the files are restored, the following prompt appears:

```
Are there any more diskettes? (y,n)
```

If you answer **y** to the above prompt, this message appears:

Next diskette ready? (y,n)

Answer y to continue the restore procedure.

When all diskettes have been restored, answer n when the prompt "Are there any more diskettes?" appears. The program returns you to the first screen. You can then type q to exit the usave utility.

HASH TABLE MAINTENANCE

Hash Table Maintenance rebuilds the hash table from scratch. Use it when you cannot access a record type by its primary key but can find it by scanning the file or by using a secondary index. This is likely to happen if you change the internal data type of a key field using Schema Maintenance and then reconfigure the database.

Hash Table Maintenance first zeroes the hash table index and then reads through the database, making an entry for each key that it finds. Start it by selecting Hash Table Maintenance from the Database Maintenance Menu. When the program is finished, you are returned to the menu handler.

EXPLICIT RELATIONSHIP MAINTENANCE

This program rebuilds the explicit relationships in the database. Use it when SQL or LST performs a "join" or when the explicit relationship functions (unisort, makeset, setsize, nextrec, prevrec) quit functioning or start returning inconsistent results.

Explicit Relationship Maintenance first zeroes the existing pointers and then reads through the database, relinking the records as necessary. This is possible because a certain amount of redundant data indicating relationships is stored. Start it by selecting Explicit Relationship Maintenance from the Database

Maintenance Menu. When the program is finished, you are returned to the menu handler.

DATABASE LOAD

The Database Load program lets you load new Unify database records and update existing ones from an ordinary ASCII file. If the ASCII file contains new key values, the data is inserted; if the file contains existing key values, fields in the existing records are updated with the ASCII file information.

You can create the ASCII file in many ways: by dumping data from an existing non-Unify application; by using a text editor to type data; by using SQL to dump data from an existing Unify database, and so on. The only requirement is that the format of the file must match the format expected by Database Load. If your file is in some other format, you can use one of the TRS-XENIX text processing utilities to transform the file into the appropriate format.

Since you will probably interact with the shell to create your ASCII file, this program is designed to be used from the shell instead of from the menu handler. This lets you use the full power of I/O redirection and pipes to load your data. However, SQL also interfaces with this program, making it easy to load data and move it within a Unify application. See "Dumping Data to TRS-XENIX Files" for information on dumping data to TRS-XENIX files from SQL and "Loading Data from TRS-XENIX Files" for information on reloading the data into the database.

Database Load expects four parameters when you call it from the shell command line. The following command line is necessary to start the program:

```
DBLOAD <database> <record type> <file> <Spec file>
```

The parameters are:

Parameter	Usage
<database>	The name of the database file to load. This is either file.db, if you are loading data into the applications database, or unify.db, if you are loading data into the data dictionary.
<record type>	The name of the database record into which you are loading the data.
<file>	The name of the ASCII input file containing the data to be loaded. If this is a dash (-), the standard input is used; thus, DBLOAD can be used at the end of a pipeline.
<spec file>	The name of another ASCII file containing the list of fields within the record type to be updated, in the order in which they are listed in the input file.

The specification file describes the format of the input file to be loaded. The file consists of a single line with an entry for each field to be updated. You need not list every field in the record type, but only the ones you want to update. However, if the record type to be updated has a key, the key must be one of the fields. The fields do not have to be in any particular order as long as the order in the specification file agrees with the order in the input file.

The specification file has this format:

field[sep]field[sep]field...[sep]

"field" is the name (either the short or long name) of the database field, and [sep] is a single character field separator indicating the end of the field. Do not use COMB fields; they

should be broken into their component parts in both the specification and input files.

The separator must not be a letter, digit, or underscore (_), since these characters are used in field names. The separator should be chosen so that it does not occur in any STRING field. A good separator is the vertical bar (|). The separator in the specification file must be the same as the separator used in the input file. Note that the last character in the specification file must be a field separator.

The input file has the format described by the specification file. Each record in the file consists of a number of fields and separators terminated with a newline character. Any extra blanks in the field are ignored. STRING fields shorter than their schema definition are padded on the right with blanks. In the example input and specification files that follow, the record's key is listed as the last field, not the first, to illustrate the free format of the files:

INPUT FILE

```
Amato|7Ø|clerk|64ØØ|95Ø.ØØ|Ø.ØØ|57ØØ
Colucci|4Ø|salesman|22ØØ|25ØØ.ØØ|3ØØØ.ØØ|67ØØ
Fiorella|4Ø|salesman|62ØØ|2ØØØ.ØØ|75Ø.ØØ|58ØØ
Sarducci|7Ø|clerk|57ØØ|8ØØ.ØØ|Ø.ØØ|68ØØ
Montesino|6Ø|engineer|13ØØ|6ØØØ.ØØ|Ø.ØØ|59ØØ
```

SPECIFICATION FILE

```
name|dept_no|job|manager|salary|commission|number|
```

Following are error messages you might see when using Database Load, with suggested actions for remedying the situations.

usage: DBLOAD <database> <record type> <file> <specfile>

You did not supply the correct number of parameters to

DBLOAD. Try again, using the four parameters listed.

Invalid format for field ffff, record nnnn

The field named ffff in record number nnnn in the input file has an invalid format. Look for an invalid date, time, or number with imbedded alpha characters. If every record has the same problem, your specification file probably does not match the input file.

Record nnnn is a duplicate key.

The record number nnnn in the input file has a key value that already exists in the database. This is not an error if you are updating existing records and not inserting new ones.

Invalid value for field, ffff, record nnnn

The field named ffff has an explicit relationship with some other field in the database, and the value for ffff in record number nnnn in the input file does not match its reference file. Either insert a record in the reference file that matches the value of the field, or change the value so that it matches an existing record.

Invalid record type: rrrr

The record type named rrrr does not exist in the database you are trying to load. You are either using the wrong database file or the wrong record type.

Can't open specification file: ffff

The specification file cannot be opened. Check to make sure that you have read privilege for the file.

Invalid field name: ffff

The field named ffff does not exist in the database. You might have misspelled it in the specification file.

Field ffff is not in record type rrrr

The field named ffff, although it exists in the database, is not in the record type rrrr. You are using either the wrong field name or the wrong record type.

SFORM -- SCREEN FORM DEVELOPMENT TOOL

The Unify screen form development tool, SFORM, maintains a database of all screen forms used in an application. SFORM stores each screen's UNITRIEVE database fields or field formats as well as their location(s) on the screen and the screen image. You can print images of the screens and listings of each screen's fields, prompts, and coordinates.

A screen can be used in one of two ways. The general purpose data entry driver, ENTER, can run behind a screen and perform simple kinds of data entry applications. See "ENTER -- Data Entry/Query by Forms" for the screen requirements necessary for using ENTER. For more complex screen control, user programs can call a set of functions for displaying or erasing screen forms and accepting or displaying data. See "Terminal I/O Functions" for a description of these.

The simplest way to create a screen form is to use the Create Default Screen Form program. If you want to design the entire screen form yourself or make changes to a screen form created through the Create Default Screen Form program, use Screen Maintenance to define every potential input and display point on the screen. Each display point is assigned a screen field name used later by the screen form functions. The following information is stored with each screen field name:

- The UNITRIEVE database field to input or display at a particular point on the screen. The same field can appear more than once under different screen field names.
- The type and display length of the field to display. If you do not specify a UNITRIEVE database field, the type and length may be used to define an editing mask for the screen field. Otherwise, the type and length are determined by the database field.
- The x and y coordinates at which the field is input and displayed on the screen.
- The prompt associated with the screen field. This is a

string of text displayed when the screen is accessed.

. The x and y coordinates of the prompt. Screen fields are numbered according to each prompt's coordinates.

Since all screen information is stored separately from programs, you can modify screens without changing or recompiling program source code. Several programs may share the same screen. The programs also have a degree of independence from the terminal and may be written without reference to actual x and y coordinates. Hardware characteristics of different terminals can also be hidden from the applications program level, enabling the same program to function on different types of terminals. These features give you programs that are easier to write, understand, and maintain.

Once a screen is defined, SFORM creates two files used by programs that reference the screen. The first, <screen name>.h, must be included (using the #include statement) in the source code of any functions that wish to refer to screen fields by name. It contains a list of screen field definitions that let the program reference screen fields by name instead of number. Screen field numbers are assigned sequentially from top to bottom, left to right, according to the prompts' x-y coordinates.

The second file, <screen name>.q, contains a binary representation of the screen format and field names. This file is used at run time to actually display the screen.

To use the screen form database maintenance programs, select the System Menu and then the SFORM Menu. Each SFORM Menu option is explained in the following sections.

SFORM Maintenance

SFORM Maintenance displays a page of screen fields to which you can add new screen fields, modify or delete screen fields, or delete entire screens. Start the program by selecting Screen Maintenance (sfmaint) from the SFORM Menu. The screen shows:

```
+---------------------------------------------------------------------+
|[sfmaint]                   UNIFY SYSTEM                             |
|                         5 OCT 1982 - 15:25                          |
|                            Screen Entry                             |
|                                                                     |
|SCREEN:########                                                      |
|                                                                     |
|LN SFIELD   DFIELD  TP LEN FX FY PROMPT                         PX PY|
|'' '''''''' ''''''''''''''''''''''''''''''''''''''''''''             |
|                    '' ''' '' '' '''''''''''''''''''''''''''''' '' ''|
|                                                                     |
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|"" """""""" """"""" "" """ "" "" """""""""""""""""""""""""""""" "" ""|
|                                                                     |
|[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE: a_                            |
|                                                                     |
+---------------------------------------------------------------------+
```

```
######## = screen form entry area
'''''''' = screen field entry area
"""""""" = screen field paging area
```

Following is an explanation of each prompt and column heading on the screen.

[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE

Lets you choose a mode for the screen form entry area. The modes are:

i -- Inquire mode, which lets you see the first page of screen fields for an existing screen form. If more than 11 screen fields have been defined, use the modify mode to view subsequent pages.

a -- Add mode, which lets you add new screen forms and associated screen fields.

m -- Modify mode, which lets you add, modify, and delete screen fields for an existing screen form.

d -- Delete mode, which lets you delete an existing screen form and all its screen fields.

When a screen is deleted, its registration with ENTER and any menu entry to select the screen is also removed.

SCREEN

A unique screen form name of up to eight characters, used to reference the screen throughout the design.

[N]ext page, [P]rev page, [A]dd line, or number:

Paging area prompt displayed after you answer the SCREEN prompt. The paging prompt lets you choose a mode for the paging area. The modes are:

n -- Displays the next page of screen fields.

p -- Displays the previous page of screen fields.

a -- Lets you add new screen fields by moving the cursor to the screen field entry area. Press <F2> to store a new screen field.

number (1-6Ø) -- Displays the page containing the indicated screen field and positions the cursor at that screen field to let you modify or delete it.

LN

A line number assigned by the system, used to reference the line if you wish to modify it.

column between LN and SFIELD (untitled)

Used for entering a command to perform an operation on the current screen field. The cursor moves up and down through the paging area in response to <F1> and <ENTER> in this column. The cursor position marks the current line. Valid commands are:

m -- Modify DFIELD, TP, LEN, FX, FY, PROMPT, PX, or PY for the current screen field.

d -- Delete the current screen field.

q -- Redisplay the paging prompt at the bottom of the screen.

SFIELD

A unique screen field name of up to eight characters, used to reference the screen field in programs you may write. For this reason, it should be unique from

database field names. However, different screen forms may use the same screen field names. The maximum number of screen fields per form is 6Ø.

DFIELD

Name of the database field to display for this screen field. If this field is left blank, you must answer the next two prompts (TP and LEN). SFORM finds the name in the data dictionary to validate it. COMB fields cannot be used directly with SFORM; enter the field components separately.

TP

The type of the database field. This is automatically answered from the data dictionary if DFIELD was specified.

LEN

The display length of the database field. This is also completed by SFORM from the data dictionary if DFIELD was specified.

FX

The x coordinate (column) of the database field on the screen form. Use a number from Ø-79.

FY

The y coordinate (row) of the database field on the screen form. Use a number from 3-2Ø. (Be sure to leave at least three blank lines at the top and bottom of a screen form for the screen heading and the prompts and error messages resulting from use of the

screen form with ENTER.)

PROMPT

Text to be displayed on the screen indicating the data to be entered and/or displayed. You may enter literal text or "escaped" characters that display the prompt in a special mode. The escape character is ~. Models II/12/16 and the DT-1 support the following special modes:

~r -- start reverse video
~s -- end reverse video

In addition, the DT-1 supports these modes:

~u -- start underline
~v -- end underline

If you start a special display mode for a prompt, be sure to end it; otherwise, your screen form will not seem proper.

PX

The x coordinate (column) of the prompt on the screen form. Use a number from Ø-79.

PY

The y coordinate (row) of the prompt on the screen form. Use a number from 3-2Ø. (Be sure to leave at least three blank lines at the top and bottom of a screen form for the screen heading and the prompts and error messages resulting from use of the screen form with ENTER.)

Test Screen

This option lets you test a screen form without writing a program to display it. Use it to check the positions of prompts and data entry points. Start it by selecting Test Screen (sfsamp) from the SFORM Menu. The screen shows:

```
[sfsamp]                        UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                                 Test Screen




SCREEN: _
```

Simply type the name of a screen form, press <ENTER>, and the screen is displayed. <F1> returns you to the menu handler.

Process Screen

This option creates the two files described in the beginning of this section for use by the applications programs. One file (<screen name>.h) is included by programs that wish to reference screen field names. The other (<screen name>.q) is read at run time by the loadscr function to display the screen and load the tables required by other SFORM functions. Start the program by selecting Process Screen (sfproc) from the SFORM Menu. The screen shows:

```
[sfproc]                        UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                                Process Screen

SCREEN: _
```

Enter a form name at the SCREEN prompt, and the form is processed, creating the .h and .q files. After the form is

processed, the screen clears. Enter another form name, or use <F1> to return to the menu handler.

Screen Reports

This program prints a copy of the screen, a list of all the screen field information, or both. The report is 120 columns in width. Start the program by selecting Screen Reports (sfrep) from the SFORM Menu. The screen shows:

```
[sfrep]                          UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                                 Screen Reports

SCREEN  _

LISTING [Y or N]:
PRINT SCREEN:
```

Following are explanations of each prompt.

SCREEN

The name of an existing screen form to print. Answering with all <ENTER> prints all screen forms in

the data dictionary.

LISTING [Y or N]

y prints a tabular report listing the attributes of each screen field. n deletes this part of the report.

PRINT SCREEN

y prints an image of the form as it would look on a screen. n deletes this part of the report.

As the report prints, a blank Screen Reports screen is redisplayed. To return to the SFORM menu, press <F1>.

Restore Screen

This option recovers screen forms inadvertently deleted from the data dictionary or reorders the prompts on a screen. If you make changes that put the prompts out of order, simply delete the screen and restore it using this program. The prompts are reordered in display sequence. The program reads the .q file in the current directory and inserts the form into the data dictionary. Start the program by selecting Restore Screen (sfrestr) from the SFORM Menu. The screen shows:

```
[sfrestr]                    UNIFY SYSTEM
                          5 OCT 1982 - 15:25
                            Restore Screen

SCREEN: _
```

Answer the SCREEN prompt with a form name, and the form is recovered from the .q file. After the form is recovered, the

screen clears. Enter another form name, or use <F1> to return to the menu handler.

When a screen form is deleted, its registration with ENTER is cancelled, and if it is also registered as a menu selection, that entry is also removed. Restore Screen only recovers the screen form. To permit the screen to be used for data entry or inquiry, the screen must be re-registered with ENTER, and if the screen is to be selected from a menu, Menu Maintenance must be run to reenter the screen selection.

Screen List

This program displays the names of all screen forms in the data dictionary. Start the program by selecting Screen List (sfslist) from the SFORM Menu. The list of screen forms is displayed. Press <ENTER> to return to the menu handler. A typical list might resemble the following:

```
[sfslist]                    UNIFY SYSTEM
                          5 OCT 1982 - 15:25
                             List Screens

 sman1ØØ
 smod1ØØ
 sitm1ØØ

 ->->_
```

Create Default Screen Form

A screen form referencing the fields in a particular record can be created, processed, and registered with ENTER in one simple step using this program. Start it by selecting Create Default Screen Form (cdsf) from the SFORM Menu. The screen shows:

```
[cdsf]                          UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                          Create Default Screen Form

RECORD: _
```

Enter the name of an existing record at the RECORD: prompt, and a default screen form for that record is generated. After the default screen form is created, processed, and registered with ENTER, the screen clears. Enter another record name, or press <F1> to return to the menu handler.

If you try to create a default screen form for a record whose fields do not fit on one screen, you see:

There is not enough room on the screen for all of the fields ->->

Pressing y at a Continue? prompt generates the default screen form using the fields that fit. n returns you to the RECORD: prompt without generating the default screen.

A final message appears before the screen clears and waits for you to enter another record name or return to the SFORM Menu:

The screen can now be used for data entry ->->

To use the default screen, enter its name at the SELECTION prompt at the bottom of any menu.

Since the screen is registered with ENTER, it can be added to a menu's list of selections via Menu Maintenance.

A default screen form can be modified or deleted using Screen Maintenance. You can also change the heading by using ENTER Screen Registration. When a screen form is deleted, its registration with ENTER is cancelled, and if it was registered as a menu selection, that entry is also removed.

The prompts are aligned in one, two, or three columns. A field's prompt is its long name or, if no long name exists, its regular or implicit name. A database field has an implicit name if it is a reference field. The implicit name consists of the field's regular name followed by the name of the field it references. The referenced field may also reference another field, adding to the implicit name. All underscores (_) are replaced with spaces.

The elements of a COMB field are given separate prompts. If a field references a COMB field, separate prompts are produced for each of its referenced elementary fields.

A default screen form's fields are positioned on the finished screen from top to bottom, in order of their appearance in the schema reports. The fields are aligned one beneath the other, starting after the longest long field name. The screen's heading is the target record's long or regular name followed by the word, "Maintenance." Any underscores (_) are changed to spaces, and appropriate letters are capitalized.

As an example, consider a default screen form created for the customer record in the Unify Tutorial Manual. The record's layout is:

RECORD/FIELD	REF	TYPE	LEN	LONG NAME
customer	2Ø			customer
*cnum		NUMERIC	5	customer_number
cname		STRING	3Ø	name
caddr		STRING	3Ø	address
ccity		STRING	2Ø	city
cstate		STRING	2	state
czip		NUMERIC	5	zip_code
cphone		STRING	14	phone_number

The default screen form created for this record would appear as:

```
+-------------------------------------------------------------------+
|[customer]                    UNIFY SYSTEM                         |
|                            5 MAY 1983 - 15:25                     |
|                          Customer Maintenance                     |
|                                                                   |
|customer number:                                                   |
|name            :                                                  |
|address         :                                                  |
|city            :                                                  |
|state           :                                                  |
|zip code        :                                                  |
|phone number    :                                                  |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|                                                                   |
|[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE _                             |
+-------------------------------------------------------------------+
```

ENTER -- DATA ENTRY/QUERY BY FORMS

The ENTER program lets you develop data entry and inquiry screens without writing a program yourself. It lets you add data to a database efficiently and provides you with a powerful and easy-to-use query interface. The interface lets you find and display the data you want without having to write a query. Once the data is found, you can update, delete, or summarize it in a report. ENTER does this by "running behind" screen forms developed with SFORM. All you need do is define the screen using the SFORM tools, and ENTER does the rest.

This chapter explains how to register an SFORM screen with ENTER (creating an "ENTER screen") and operate the registered screen. ENTER screens interface automatically with the menu handler, so you can apply program-level security to them. The selection file resulting from an ENTER query by forms request can be used by ENTER itself, by the listing processor, or by a custom program written in a host language.

When developing a screen for use by ENTER, you must follow a few rules. These rules determine the order in which fields are accepted and validated on the screen and whether or not ENTER can drive the screen at all. It is possible to define screens with SFORM that ENTER cannot drive. In this case, a host language program can be used to drive the screen.

An ENTER screen deals with a single "target" record type (the one being added, deleted, modified, or inquired upon). Fields from other record types ("secondary" records) may appear on the screen to create "views" of the database but are for display only. There must be explicit relationships (defined through Schema Maintenance) between the target and any secondary records for ENTER to work properly. The following points summarize the basic rules for screens you might want to drive with ENTER.

- Only the specified "target" record is manipulated by an ENTER screen. The target record is defined in ENTER Screen Registration. Fields from other record types are for display only. Only target records can be added or deleted. Fields

from other records display their contents only if an earlier field on the screen was an explicit reference field from the target record to that record type. For example, in the Unify Tutorial Manual, manufacturer name is displayed on the model screen because it is preceded by the manufacturer id number, an explicit reference from the model to the manufacturer.

- If an entire COMB field is an explicit reference to another record, ENTER validates all the elements of the combination before trying to store the COMB field in the database. If portions of the combination reference another record, each portion is validated in turn. If no references are defined on a COMB field or its pieces, each piece is accepted and stored in the order in which it is defined in the COMB field. Because of this rule, it is usually better to put all the parts of a COMB field together on the screen to keep the cursor from "jumping around."

- In Add mode, the record key is accepted before any other field may be entered. If the key is a COMB, all portions must be accepted and validated. It is usually best to put the record key at the top of the screen.

- All screen fields must have **both** the database field coordinates, FX and FY, and the prompt coordinates, PX and PY, even if there is no database field or prompt. ENTER uses the FX-FY combination when erasing data from the screens of terminals with no protected fields. The PX-PY combination is used to order the fields on the screen and determine the way the cursor moves from one field to the next. Leaving FX-FY at Ø erases the top line of the screen when it is cleared. Leaving PX-PY at Ø results in erratic cursor movement from one field to the next.

- ENTER uses the last three lines of the screen to display information, prompts, and error messages. Do not use lines 21, 22, or 23 to display your data.

Registering an ENTER Screen

Before ENTER can drive an SFORM screen, it must be registered. The registration process tells the menu handler the heading to display when the ENTER screen is running as well as the target record. The target record must be specified, since fields from several records might be on the screen. After an SFORM screen is registered, you can place it on menus wherever desired.

Start the program by selecting ENTER Screen Registration (entmnt) from the System Menu. The screen shows:

```
[entmnt]                        UNIFY SYSTEM
                              5 OCT 1982 - 15:25
                          ENTER Screen Registration

SCREEN NAME:

TARGET RECORD:

HEADING:

REPORT SCRIPT: 1:
               2:
               3:
               4:




[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE _
```

One SFORM screen at a time can be registered with ENTER. Following are explanations of the prompts.

[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE

Lets you choose a mode for the program. The modes are:

i -- Inquire mode, which lets you see a screen form already registered with ENTER.

a -- Add mode, which lets you register screen forms with ENTER.

m -- Modify mode, which lets you modify TARGET RECORD, HEADING, and REPORT SCRIPTs 1-4 for a screen form already registered with ENTER.

d -- Delete mode, which lets you remove the ENTER registration for a screen form.

SCREEN NAME

The name of the screen form to register.

TARGET RECORD

The name of the record type to be maintained by the screen form (the one added, modified, and so on). You must specify the target record, since fields from many different records might appear on the screen form.

HEADING

A string of up to 34 characters to be displayed on the third line of the screen when ENTER is running. The heading is also displayed on any menus on which the screen form appears. You can enter literal text or "escaped" characters that display the heading in a special mode. The escape character is the tilde (~).

Models II/12/16 and the DT-1 support the following special modes:

~r -- start reverse video
~s -- end reverse video

In addition, the DT-1 also supports these modes:

~u -- start underline
~v -- end underline

If you start a special display mode for a heading, be sure to end it; otherwise, your screen form will not seem proper.

REPORT SCRIPT

Each prompt, numbered 1-4, lets you enter an eight-character report script name to associate with the screen. Up to four reports can be associated with a screen. The report script is a TRS-XENIX file that contains commands to the reporting part of the Listing Processor about formatting the report. See "Listing Processor" for more information.

If you make any change or addition in ENTER Screen Registration to a screen form that appears as an option on a menu, return to the menu **prior to** the menu on which this screen form appears as an option, and then reselect the menu containing the screen. This procedure is necessary to reflect the changes you made in ENTER Screen Registration.

Using ENTER Screens

An ENTER screen has four modes of operation -- Add, Inquire, Modify, and Delete. Some of the modes may or may not be available to you, depending upon your access privileges. In Add mode, the primary key of the target record is accepted first. If the primary key is composed of multiple fields, all fields must appear on the screen. Once you enter the key, you may then add data for the remaining fields of the target record appearing on the screen.

In the other three modes, you can use ENTER's query by forms capability to find records by filling in any desired prompt(s) on the screen. You can then examine the set of records satisfying the query one-by-one and update or delete them. The following sections describe the two major ways of using ENTER -- data entry and query by forms.

ENTER DATA ENTRY

Data entry is the process of adding new data to a database or modifying existing data. This section describes the Add and Modify modes. Because of the nature of ENTER, its use is best understood through examples rather than a reference description. For this reason, we recommend that you refer to the Unify Tutorial Manual for examples of using an ENTER screen.

In general, ENTER screens work as follows. In Add mode, the cursor moves first to the key field. You must enter the key before the cursor moves to the next prompt. If the key is a COMB field with several parts, you must enter all its parts. Once you specify the key, the record is immediately added to the database and locked so that users on other terminals may not modify or delete it until all the remaining fields of the target record have been added and are stored by pressing <F2>.

In Modify mode, the cursor also moves first to the key field. However, here you can use query by forms to locate the desired record(s). If someone else is using the record

(modifying or deleting it) on another terminal, an error message is displayed when you try to modify it, and you are not allowed to update the record. When the other user finishes, you can try again. Once you locate the desired record, the screen acts in the same way as it does in Add mode.

Once a record is added or found, the cursor moves to the next prompt. From there, pressing <ENTER> moves the cursor down the screen, and <F1> moves it up the screen. When you reach the last prompt, <ENTER> moves the cursor back to the top. <F1> at the top prompt moves the cursor back to the bottom. Fields from records other than the target record are skipped, since they are for display only. <F2> stores the entries and then erases the data on the screen to prepare for the next operation. Pressing <F1> at the first prompt redisplays the [I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE prompt at the bottom of the screen. <CTRL><X> at any time immediately returns you to the menu handler.

To change the data displayed at a prompt, press <ENTER> until the cursor is at the prompt. Enter the new data over the old, and press <ENTER>. All fields in the target record, including the key field, can be modified this way. If the new data is accepted, it is stored in the database, and the cursor moves to the next prompt. If the data is not accepted, an error message is displayed at the bottom of the screen, and the database is not updated. Press <ENTER> to cancel the error message, and try again.

If you start to enter data and change your mind, <F1> cancels the entry without changing the database. The old data is redisplayed to verify that it was not changed.

To change the data in a secondary record, define another ENTER screen with that secondary record as the target, or use a host language program to drive the screen.

ENTER QUERY BY FORMS

Query by forms is the process of finding a record or set of records to examine, modify, or delete. This section describes

how to locate those record(s). As with data entry, query by forms is best illustrated in the context of a specific example, so we recommend that you refer to the Unify Tutorial Manual for a query by forms example.

Once you have selected Inquire, Modify, or Delete mode, the following prompt appears:

```
Begin search [CTRL E], Clear field [CTRL Z], Exit [CTRL X]
```

These are the default control keys for these functions. You can change them by editing your termcap file. These control characters perform the following functions:

<CTRL><E> -- Starts the search with the selection options you entered. You can also press <ENTER> until you pass the last screen field, and the search begins.

<CTRL><Z> -- Clears the current selection specification. This is used when you enter a field on the screen and later decide not to search on that field.

<CTRL><X> -- Terminates the current ENTER screen and returns you to the menu handler. <CTRL><X> may be used anytime. This is a quick way of exiting an ENTER screen.

The simplest query you can perform is to find a record by its primary key. Simply enter the key value, and the record is found immediately.

ENTER provides two other basic types of queries -- exact matching, in which you type exactly the data you want the

selected records to contain; and inexact (or generic) matching, in which you provide special characters that expand into patterns during the matching process. Examples of inexact matches are numeric and date ranges (such as numbers from 1-1ØØ or the dates 3/1/82 to 3/31/82) or substring matching (such as all strings containing the letters "Smith").

To specify an exact match, fill in the field(s) on the screen with the data you want to see. To specify an inexact match on string fields, use a set of special characters that are expanded into patterns to be matched. The special characters and their meanings are:

? -- The "wild character." The question mark matches any single character. If you wanted to find all the Smiths and did not know if they were spelled "Smith" or "Smyth," you could type **Sm?th**.

* -- The "wild string." The asterisk matches any string of characters of any length, including zero length strings (also called "null strings"). ENTER automatically adds * to the end of all string specifications.

[...] -- A set of characters that defines a "character class." The character class matches any single character that is a member of the class. Ranges of characters are specified by separating two characters by a hyphen (-). For example, all uppercase letters could be represented by the class:

[ABCDEFGHIJKLMNOPQRSTUVWXYZ]

or more conveniently as [A-Z]. All letters, upper- and lowercase together, could be represented by [a-zA-Z].

Inexact matches on non-string fields can be constructed by the following set of expressions. In the following, "f1" and "f2" are the field values you supply.

>f1 -- The "greater than" operation. All fields with values greater than the entered value are found.
<f1 -- The "less than" operation. All fields with values less than the entered value are found.

!f1 -- The logical "not" operation. All fields that do not match the entered value are found.

f1-f2 -- The "range" operation. All field values that match or are between the entered values are found. This is the same as >=f1 AND <=f2.

!f1-f2 -- Finds all field values outside the range of entered values. This is the same as <f1 OR >f2.

Any fields can be completed using the above characters. All qualifications are added together to produce a query. Once a set of records has been selected to meet the criteria, the first record in the set is displayed with a prompt giving you a selection of available operations. The available operations are:

[N]EXT, [P]REVIOUS, [M]ODIFY, [D]ELETE, [R]EPORT, [S]TOP

The options actually displayed are chosen from this set. [N]EXT, [P]REVIOUS, and [S]TOP are displayed for all modes. [M]ODIFY and [D]ELETE are displayed only in Modify or Delete modes. If report script names are specified during ENTER Screen Registration, the [R]EPORT option is displayed when in Inquire mode. The responses are:

n -- Displays the next record in the set of retrieved records. <ENTER> also performs the same function.

p -- Displays the previous record in the set of retrieved records.

m -- Moves the cursor to the first screen field so you can modify the data in the record. If someone else is modifying or deleting this record, you cannot modify it at the same time.

d -- Deletes the record from the database. If someone else is modifying or deleting this record, you cannot delete it at the same time.

r -- Uses the set of records selected during query to produce a report. The format of the report is taken from a report script specified for the screen in ENTER Screen Registration. If more than one report script is associated with the screen, a subsequent prompt is displayed requesting the number of the report to print.

s -- Clears the current set of retrieved records and clears the screen of data so that you can perform a different query. <F1> also performs the same function.

searched: nnnnnn -- Shows the number of records examined to satisfy the query.

Selected: nnnnnn -- Shows the number of records selected.

current: nnnnnn -- Shows which record within the set is currently displayed.

Customizing ENTER

DISREGARD THIS SECTION UNLESS OPERATING WITH THE TRS-XENIX DEVELOPMENT SYSTEM.

Though ENTER lets you edit database field formats and explicit relationships, you might occasionally need to perform more specialized edits or additional database processing -- for example, edits for ranges of values; comparison with a set of valid strings or values; comparison with other database fields; assigning default values (such as a sequential key number or today's date); or even skipping a field based on the contents of another field.

For this reason, the ENTER archive is included with Unify, in the lib/enter.a file. ENTER provides a means of calling user defined functions to customize ENTER for your application. This makes it easier to develop a custom screen -- the basic program logic is already implemented; you simply add the special functions you want to perform.

Your custom version of ENTER can contain special processing for many different screen forms and still work with all other ENTER screens in the standard way. If there is not enough space for all your screens, you can even create multiple ENTER executables with different names. This section explains the usage of user defined functions and gives examples of customized ENTER screens with instructions on creating them.

THEORY OF OPERATION

Five different kinds of user functions can be included in a custom version of ENTER. The functions are: screen form initialization; screen field preprocessing; input; post processing; and screen form termination. These functions can reference global variables indicating the current screen name and processing mode (Add, Inquire, Modify, or Delete).

Tables that you define tell ENTER which functions to use for each custom screen form. Screen forms not in the table are processed in the standard way.

The screen initialization and termination functions are called by ENTER's main routine, and the preprocessing, input, and post processing functions are called by the ENTER screen field input routine. This routine is called e_inbuf, and it obtains each screen field data element from the user in Add and Modify modes. The calls to user defined functions are incorporated in this function. When ENTER is in query by forms mode, you have no control over what happens.

When e_inbuf obtains a screen field data element from the user, you can execute a function from one of three points: **before** the cursor moves to get the input (the preprocessing function); as the cursor gets the input (the input function); and **after** the input is entered (the post processing function). Any function you specify is always called, regardless of the user response. The only exception is when the preprocessing function returns a nonzero value, which results in an immediate return. You may specify any combination of preprocessing, input, and post processing functions for any field.

You specify the function to call at a particular point by initializing tables including the names of the screen forms, screen fields, and function. The table definitions are in the include file, edits.h. This file also contains external references to variables that define the current operational mode, scrmode, and the name of the current screen form, _savscr. The first table, SFTABLE, contains a list of screen fields that have associated functions. The table has the following format:

```
struct sftable {
                int ssfld,      /* the screen field number  */
                (*prefunc)(),   /* preprocessing function   */
                (*inpfunc)(),   /* input function           */
                (*postfunc)()   /* post processing function */
               };
#define SFTABLE struct sftable
```

You can use as many of these tables as you want, with any names you choose, one for each screen form that you want to

customize. Each entry in the table corresponds to a screen field and contains the screen field number and the addresses of the preprocessing, input, and post processing functions. If you do not want to use one of the functions, put a Ø in the corresonding slot in the table. The last entry in the table must have a -1 in the ssfld element and Øs for each processing function.

The second table, SCRTABLE, tells ENTER which SFTABLE to use for a particular screen form. This table has the following format:

```
struct scrtable {
                char *sname;       /* the screen form name     */
                int (*inifunc)(), /* initialization function */
                    (*trmfunc)(); /* termination function     */
                SFTABLE *scrdf;    /* processing function      */
                };
#define SCRTABLE struct scrtable
```

There is only one SCRTABLE, and you must define it with the name, screentab. ENTER is compiled with an external reference to this name, and it is undefined if you do not use it when setting up your tables. Each entry in the table contains the name of a screen form, the addresses of the initialization and termination functions for the form, and the address of the SFTABLE containing the functions to customize this form. You must specify the screen name and the SFTABLE address, but the function addresses can be set to Ø if you do not want to do anything. The last entry in this table must have a Ø in every slot.

When ENTER starts, it searches screentab to see if there are any custom functions for the screen form. It compares the current screen name with the names in screentab. If it finds a match, ENTER uses the indicated SFTABLE to find the user functions to execute. Following are pseudo-code versions of the ENTER control routine and e_inbuf that indicate where and how the user functions are called, followed by reference-style descriptions of the generic user functions.

```
enter ()
{
    display screen ();
    if (user initialization function exists for this screen)
        user initialization function ();

    while ((operational mode = priamd (1)) != F1)
    {
        switch (operational mode)
        {
            case ADD:
                add mode ();
            case MODIFY:
                modify mode ();
            case INQUIRE:
                inquire mode ();
            case DELETE:
                delete mode ();
        }
    }
    if (user termination function exists for this screen)
       user termination function ();
}
```

```
e_inbuf (screen field, buf)
{
  while (1)
  {
    if (preprocessing function exists for screen field)
    {
      error status = preprocessing function (screen field, buf);
      if (error status != Ø)
        return (error status);
    }

    if (input function exists for screen field)
      error status = input function (screen field, buf);
    else
      error status = system input function (screen field, buf);

    if (post processing function exists for screen field)
    {
      if (post processing function(screen field,buf,error status) == Ø)
        return (error status);
    }
    else
      return (error status);
  }
}
```

INIFUNC **INIFUNC**

Name: inifunc -- perform screen form initialization

Syntax: inifunc ()

Description: ENTER calls this function after the screen form is displayed, but before the INQUIRE, ADD, MODIFY, DELETE prompt is displayed. Since the standard ENTER is executed when Unify starts and is connected to the menu handler with a pipe, any tables you read in or records you make current remain that way across invocations. This is only significant if you want to allocate more memory using sbrk(2) or similar system calls. You need only do it once, not every time a new screen is called. If your version of ENTER is named something else, this does not apply, since the program is executed for each different screen.

Typical uses for this function would be opening or creating temporary files, accessing parameter records in the database, or asking questions of the user. The name of the specific initialization function for each screen form is put into the screentab SCRTABLE.

Returns: The return status code is not checked by ENTER.

TRMFUNC TRMFUNC

Name: trmfunc -- perform screen form termination

Syntax: trmfunc ()

Description: ENTER calls this function after you enter <F1> at the INQUIRE, ADD, MODIFY, DELETE prompt or after you "kill" the ENTER screen with <CTRL><X>. Typical uses for this function would be closing or unlinking temporary files, performing additional editing, or database processing. The name of the specific termination function for each screen form is put into the screentab SCRTABLE.

Returns: The return status code is not checked by ENTER.

PREFUNC PREFUNC

Name: prefunc -- screen field preprocessing

Syntax:
```
prefunc (sfield, buf)
int sfield;
char *buf;
```

Description: If specified, ENTER always calls this function before the indicated screen field is input. The parameters are the screen field number and the address of a buffer into which data should be placed. Typical uses include skipping input of the current field or calculating and displaying a field. The name of each specific preprocessing function is put into an entry in an SFTABLE structure that you declare.

Returns:
- Ø -- continue with input processing
- -2 -- return to the previous field without inputting this one
- -3 -- go on to the next field without inputting this one

INPFUNC INPFUNC

Name: inpfunc -- screen field input processing

Syntax:
```
inpfunc (sfield, buf)
int sfield;
char *buf;
```

Description: If specified, ENTER calls this function instead of the standard ENTER input routine. The parameters passed to it are the screen field number and the address of the buffer into which data should be placed. Typical uses are assigning default values for key fields (such as a sequential number) if the user presses <ENTER> or masking the input field in some way (such as for a telephone number with parentheses around the area code). The name of each specific input processing function is put into an entry in an SFTABLE structure that you declare.

The post processing function can also be used to assign default values for non-key fields only. The standard ENTER input routine treats key fields as "must enter" fields and does not allow a carriage return (<ENTER>).

Returns:
```
 Ø -- go on to the next field, data was entered
-2 -- return to the previous field, no data was entered
-3 -- go on to the next field, no data was entered
```

POSTFUNC **POSTFUNC**

Name: postfunc -- screen field post processing

Syntax:
```
postfunc (sfield, buf, ier)
int sfield, ier;
char *buf;
```

Description: If specified, ENTER calls this function after the indicated screen field is input. The function is always called, even if you pressed only <ENTER> or <F1>. The parameters are the screen field number, the address of the buffer containing any data you entered, and the error status code returned from the input function. The most typical use for this function is performing additional editing on the value you entered. If the function returns a Ø, e_inbuf returns to ENTER with the error status code of the input function. Otherwise, e_inbuf starts again with the preprocessing function.

You could also use this function to perform additional database processing, such as inserting or deleting records in other files, updating other database fields, or even executing another user program. The name of each post processing function is put into an entry in an SFTABLE structure that you declare.

Returns: Ø -- allright to continue
non-zero -- start the input process again for this screen field

CUSTOM ENTER EXAMPLES

This section contains a few examples of the customization functions described in the previous section. The source code for these examples and the resulting custom version of ENTER is in the tutorial directory included with this package.

The schema used is the one presented in the Unify Tutorial Manual, with manufacturers, models, items, customers, and orders. To illustrate automatic key generation, another record type has been added -- the database parameter record, "node." This record stores the current highest key value of manufacturers, customers, and items, so that the next sequential number can be easily found. Only one node record exists, with the key of "A." This key is coded into programs that need to access the node record. You will see how this is used later. The schema design used is:

RECORD/FIELD	REF	TYPE	LEN	LONG NAME
manf	1Ø			manufacturer
*mano		NUMERIC	4	number
mname		STRING	35	name
madd		STRING	35	address
mcity		STRING	2Ø	city
mstate		STRING	2	state
mzip		NUMERIC	5	zip code
model	5Ø			model
*mokey		COMB		model_key
monum		NUMERIC	7	model_num
momano	mano	NUMERIC	4	manufacturer_num
mdes		STRING	3Ø	description
item	1ØØ			inventory_item
*sno		NUMERIC	9	serial_number
imodel	mokey	COMB		model_id
iad		DATE		acquisition_date
isal		AMOUNT	5	sales_price
iorder	onum	NUMERIC	9	order_number
ipamt		AMOUNT	5	purchase_price

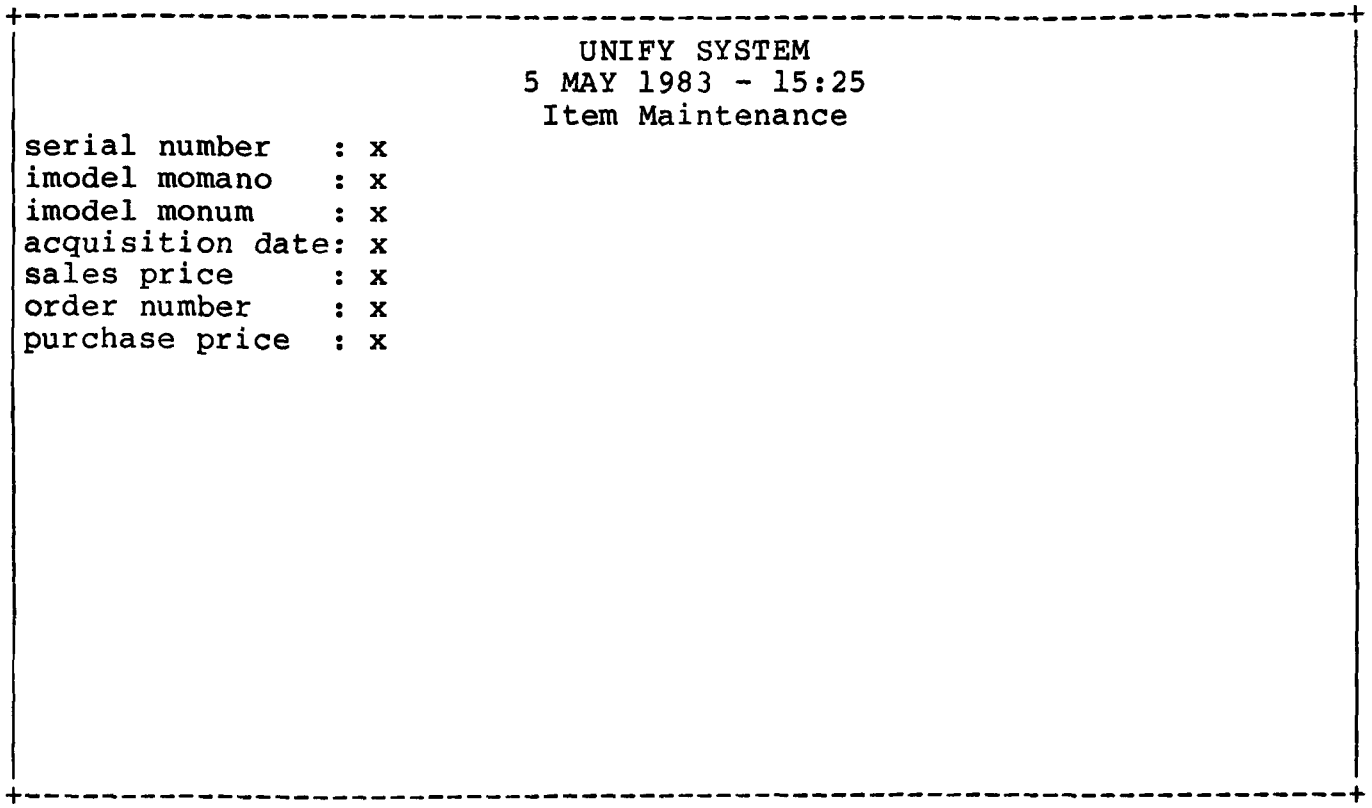
```
customer          1Ø                                  customer
    *cnum                          NUMERIC    5      customer_number
     cname                         STRING     3Ø     name
     caddr                         STRING     3Ø     address
     ccity                         STRING     2Ø     city
     cstate                        STRING      2     state
     czip                          NUMERIC    5      zip_code
     cphone                        STRING     14     phone_number

orders            1ØØ                                 orders
    *onum                          NUMERIC    9      order_number
     odate                         DATE               date_ordered
     ocust           cnum          NUMERIC    5      customer_number

node               1                                  node
    *nkey                          STRING     1
     nmanfseq                      NUMERIC    4      manf_sequence_no
     nitemseq                      NUMERIC    9      item_sequence_no
     ncustseq                      NUMERIC    5      cust_sequence_no
```

Suppose that you want to customize two ENTER screens -- one for the customer record and one for the item record. The basic screen forms used were generated using the Create Default Screen Form option on the SFORM Menu. The two forms are laid out as follows:

```
                          UNIFY SYSTEM
                        5 MAY 1983 - 15:25
                       Customer Maintenance
customer number: x
name           : x
address        : x
city           : x
state          : x
zip code       : x
phone number   : x
```

SCREEN: customer

SCREEN FIELD	DATA FIELD	TYPE	LEN	FX	FY	PROMPT	PX	PY
custom01	cnum	N	5	18	4	customer number:	1	4
custom02	cname	S	30	18	5	name :	1	5
custom03	caddr	S	30	18	6	address :	1	6
custom04	ccity	S	20	18	7	city :	1	7
custom05	cstate	S	2	18	8	state :	1	8
custom06	czip	N	5	18	9	zip code :	1	9
custom07	cphone	S	14	18	10	phone number :	1	10



SCREEN: item

SCREEN FIELD	DATA FIELD	TYPE	LEN	FX	FY	PROMPT	PX	PY
item01	sno	N	9	19	4	serial number :	1	4
item02	imodel_momano	N	4	19	5	imodel momano :	1	5
item03	imodel_monum	N	7	19	6	imodel monum :	1	6
item04	iad	D	2	19	7	acquisition date:	1	7
item05	isal	A	5	19	8	sales price :	1	8
item06	iorder	N	9	19	9	order number :	1	9
item07	ipamt	A	5	19	10	purchase price :	1	10

Custom ENTER Functions

This section describes the custom functions for the two sample screen forms and points out some unobvious details. For the customer screen, you want to assign the next sequential key value if the user presses <ENTER> for the key value in Add mode. You also want to edit the state code in both Add and Modify modes to make sure that it conforms with the valid two-character U.S. Postal Service state codes. This requires an input function on the customØ1 field and a post processing function on the customØ5 field.

For the item screen, you want to assign the next sequential key value if the user presses <ENTER>. This requires an input function for the itemØ1 field. Then, for the acquisition date, you want to assign a default value of today's date in Add mode if the user does not enter the date. You want to edit the date so that it is less than or equal to the system date if the user enters a date. This requires a post processing function for the itemØ4 field.

Finally, you want to skip the purchase price field if the sales price field is Ø and edit the purchase price to make sure that it is less than or equal to the sales price. This requires several functions -- a preprocessing function to skip the itemØ7 field and a post processing function to edit it after it is entered. Since valid values for the purchase price now depend on data in the sales price field, you must also check that a change to the sales price has not made the purchase price invalid. This is done with a post processing edit on either screen field itemØ5 or itemØ1. For illustration, the post processing function on the itemØ1 screen field is used. This shows how you can re-edit the record after the user has completed it.

The first step in customizing screen forms is to create the function tables that tell ENTER which functions to call. You need two SFTABLE structures: one for the customer screen form and one for the item screen form. You also need the standard SCRTABLE screentab to relate the screen forms to their corresponding SFTABLE structures. The required function tables' definitions are:

```
#include "../../include/edits.h"

int gcuskey(), ed_state ();

#include "../../def/customer.h"
SFTABLE custedits[] = {

/* screen field */      /* pre */       /* input */     /* post */

custom01,               0,              gcuskey,        0,
custom05,               0,              0,              ed_state,
-1,                     0,              0,              0 };

int ed_date(), gitmkey (), ed_intf (), cksal(), ed_pamt(), initeds();

#include "../../def/item.h"
SFTABLE itemedits[] = {

/* screen field */      /* pre */       /* input */     /* post */

item01,                 0,              gitmkey,        ed_intf,
item04,                 0,              0,              ed_date,
item07,                 cksal,          0,              ed_pamt,
-1,                     0,              0,              0 };


SCRTABLE screentab[] = {
/* screen name */       /* pre */       /* post */      /* edits */

"customer",             initeds,        0,              custedits,
"item",                 initeds,        0,              itemedits,
0,                      0,              0,              0 };
```

Choose any function names and SFTABLE structure names you like. The only requirement is that you name the SCRTABLE structure screentab.

Note some of the editing and processing functions. The first one is the initialization function, initeds. In this case, the same initialization function is used for both screen forms, but you could use a different function for each form if you

wanted. This function only accesses the node record, since other editing functions need to use fields from this record. This function's code is:

```
$include "../../def/file.h"

initeds ()
{
  if (access (node, "A"))
  {
    prtmsg (Ø,23,"Unable to access database parameter record");
    prtmsg (Ø,23,"This program will exit now");
    exit ();
  }
}
```

The next function is the input routine for getting the customer's ID number. If the user types a number, that number is used for the key. If the user presses <ENTER> instead, the function assigns a sequential key value. The next sequential number is obtained from the node field, ncustseq. The number in the node is the smallest key value likely to be available.

Two users trying to add new customers at the same time could get the same ID number. In this case, one would succeed, and the other would see the message, "Record already exists," since the addrec function insures that two different records with the same key cannot exist. In this case, the user could simply try again.

Note the use of the global variable, scrmode. This variable is externally referenced in edits.h, and it indicates the current mode of ENTER. Symbols are also defined in edits.h that you can use to test the current value of scrmode. The code for this function is:

```
$include "../../def/file.h"
#include "../../def/item.h"
#include "../../../include/edits.h"
```

```
short gotckey = Ø;

gcuskey (sfield, buf)
int sfield;
char *buf;
{
    int ier;
    long ltemp, lrec;

    if (scrmode != ADD)
        return inbuf (sfield, buf);

    if ((ier = inbuf(sfield, buf)) == Ø)
    {
        gotckey = 1;
        return (Ø);
    }
    else
    {
        if (ier == -3)
        {
            if (gotckey)
                return (-3);
            gfield (ncustseq, &ltemp);
            loc (customer, &lrec);
            while (access (customer, &ltemp) == Ø)
                ltemp++; /* find the next available
                            customer number */
            rloc (customer, lrec);
            pfield (ncustseq, &ltemp);
            outbuf (sfield, &ltemp);
            *(long *)buf = ltemp;
            gotckey = 1;
            return (Ø);
        }
        else
        {
            gotckey = Ø;  /* code was F1 */
            return (ier);
        }
    }
}
```

If Add is not the current mode, the routine returns the value of inbuf for the screen field. Now that you know you are in Add mode, inbuf is used to get the value from the user. If data is entered, a global flag is set to indicate that the customer key has been determined, and a Ø status is returned. If the user presses <ENTER> (-3), you need to generate the next sequential key. If the key value has already been determined, you simply return a -3. This could happen when the user loops back to the top of the screen after adding the record and then presses <ENTER> again. In this case, you do not want to generate another key value.

To generate the key, you first get the candidate value from the ncustseq field in the node record, and then save the address of the current record, if any. ENTER may think something is current, and you must keep everything the way you found it. Then, you attempt to access records using this value, incrementing it until you find a record that does not exist. You next restore the current record and again store the key value in the node record. Finally, you display the key value on the screen, store the generated key in the buffer (just as if the user had entered it), set the flag to indicate that the key has been determined, and return a Ø status, meaning that something was entered. ENTER places the buffer value in the record, exactly as if the user had entered it.

If the user enters <F1> (-2 status), you must clear the "key determined" flag and return the status to ENTER. At this point, the record is not accepted, because no key was entered and <ENTER> was not pressed to generate the key. ENTER clears the screen and lets the user reenter a record.

The edit for validating state codes is easier. The post processing function must note the value entered by the user and compare it to a table of valid codes. (The table in this sample does not include all possible codes.) If the entry matches one of the valid values, the routine returns a Ø status. Otherwise, it displays an error message, redisplays the current value of the field, and returns a -1 status. This makes e_inbuf prompt again for input from the user. Note that you still want to edit the

field if the user only presses <ENTER> (status code -3). This requires getting the field from the database to see its current value. This edit lets a null or Ø value pass. STRING fields in the database are set to all Øs when the record is added.

```
$include "../../def/file.h"
char *statecds[] = {"  ", "CA", "OR", "NY", "HI", "NH", "WA", Ø};

ed_state (sfield, buf, ier)
int sfield, ier;
char *buf;
{
    char xstate[3];

    if (ier == -3)
    {
        cfill (Ø, xstate, 3);
        gfield (cstate, xstate);
        if (valstr (xstate, statecds) == 1 || xstate[Ø] == Ø)
                return (Ø);
    }
    if (ier == -2)              /* don't edit on F1 */
        return (Ø);
    buf[2] = Ø;                 /* null terminate the buffer */
    if (valstr (buf, statecds) == 1)
        return (Ø);             /* valid state code entered */
    else
    {
        prtmsg (1, 23, "Invalid state code");
        output (sfield);        /* redisplay current value */
        return (-1);
    }
}
```

The function gitmkey for the item screen form is almost identical to that for the customer screen form, so it is not discussed here. The date edit routine ed_date, is an example of a post processing function being used to assign a default value. Here is the source code for this function:

```
$include "../../def/file.h"
#include "../../../include/edits.h"


ed_date (sfld, buf, ier)
int sfld, ier;
char *buf;
{
    short jday;

    if (ier != Ø)
        gfield (iad, &jday);
    else
        jday = *(short *)buf;

    if (ier == -3 && scrmode == ADD && jday == -32768)
    {
        jday = curdate ();
        outbuf (sfld, &jday);
        pfield (iad, &jday);
        return (Ø);
    }

    if (ier != -2 && jday > curdate ())
    {
        prtmsg (1, 23, "The date must be today or earlier");
        output (sfld);
        return (-1);
    }
    return (Ø);
}
```

In this function, if the user enters data, it is put into a temporary variable; otherwise, the value in the database is retrieved. If the user presses <ENTER> (status code -3) in Add mode and the date is null (such as -32768), you then assign a default value of today's date. You must store this in the database, since the <ENTER> value (-3) is returned to ENTER. ENTER stores the buffer in the database only when it receives a Ø status from the input routine.

If the user presses <ENTER> or enters a valid date, you check to make sure that it is less than or equal to today. If not, you print an error message, redisplay the current date value, and return a -1 status. This causes the field to be prompted again.

One use for a preprocessing function can be seen in the cksal function. This function skips entry of the purchase price field if the sales price is equal to Ø. It gets the value of the sales price field and, if it is Ø, returns the <ENTER> value (-3) to the input routine. This causes the input routine to immediately exit to ENTER with the status code. The result is exactly the same as if the user had pressed <ENTER> for the field, except that the cursor never moves to the field. The code for this function is:

```
$include "../../def/file.h"

cksal (sfld, buf)
int sfld;
char *buf;
{
    long xisal;

    gfield (isal, &xisal);
    if (xisal ==Ø)
        return (-3);  /* skip over the paid amount field */
    else
        return (Ø);
}
```

If the sales price field is not Ø, the user can input the purchase price. After accepting the response, the post processing function ed_pamt is called. This function compares the sales price with the purchase price and makes sure that it is less than or equal to the sales price.

```
$include "../../def/file.h"
#include "../../include/edits.h"

ed_pamt (sfield, buf, ier)
int sfield, ier;
char *buf;
{
    long xisal, *ltmp, xipamt;

    gfield (isal, &xisal);
    if (ier != Ø)
        gfield (ipamt, &xipamt);
    else
    {
        ltmp = (long *)buf; /* some compilers have trouble   */
        xipamt = *ltmp;   /* with long assignments like this */
    }
    if (xisal < xipamt)
    {
        prtmsg (1, 23, "The purchase price must be less than or
                        equal to the sales price");
        output (sfield); /* redisplay the current data */
        return (-1);
    }
    return (Ø);
}
```

This is good for editing the purchase price field when the user changes it, but what if the user again changes the sales price, making it smaller than the purchase price? You could either put a variation of this post processing function on the itemØ5 field to take care of it or re-edit the record when the user tries to clear the screen. This sample uses the latter approach to illustrate the point.

The user can clear the screen by pressing <F2> at the first field, itemØ1 on this screen form. Thus, a post processing function on this field can check any inter-field dependancies one last time when the user presses <F2>. The function is called ed_intf, and the source code for it follows.

```
$include "../../def/file.h"
#include "../../def/item.h"

extern short gotikey;

ed_intf (sfield, buf, ier)
int sfield, ier;
char *buf;
{
    long xipamt;

    if (ier == -2)
    {
        gfield (ipamt, &xipamt);
        if (ed_pamt (itemØ7, &xipamt, Ø) != Ø)
            return (-1);
        gotikey = Ø;
    }
    return (Ø);
}
```

This function is very straightforward. If the user enters <F2> (status code -2), the function gets the purchase price from the database and calls the purchase price post processing edit function. If it does not pass the edit, -1 is returned to the system input routine, causing it to reprompt for the field. The user could go to either the sales price or the purchase price field and correct the problem. Note that the "key determined" flag is cleared by this routine, not by gitmkey. A nonzero status from the post processing function can cause the key to be reprompted, and you still need to know that the key has already been determined.

Loading ENTER

This section explains how to load your functions with ENTER once you have written them. Included with your Unify release is a shell command file, enter.ld. It is in the unify executable

directory, bin, and contains the commands needed to load ENTER using the ENTER archive in the unify/lib directory. It takes a single argument -- the path name of the archive containing your custom functions.

Before you can execute enter.ld, you must compile your functions and place them in an archive. To do this for the tutorial example, you would change the directory to tutorial/src/sfe and compile all the programs using the ucc command:

```
ucc -c -Ø *.c <ENTER>
```

Then, you would create the archive and remove the .o files with the following commands:

```
ar crv xxxx.a *.o <ENTER>
rm *.o <ENTER>
```

The sfedit.a archive already exists in this directory. The order of the functions in the archive is most important. The function table definitions must occur in the first .o in the archive, and all functions in the archive must be referenced in the function table definitions. To examine the order of functions in the sfedit.a archive, enter the command:

```
/order sfedit.a
```

In the tutorial example, all the function tables are defined in the source file, initeds.c, and initeds.o is the first function in the archive.

If the archive is not in the correct order, your functions will not load properly, and undefined symbols will occur at the end of the loading process. For your custom ENTER screens, you can try to correct the problem using the command:

ar crv xxxx.a `lorder *.o | tsort` <ENTER>

as explained in the TRS-XENIX System Reference Manual in lorder(1). This command reorders your archive but might not always work. If you still get undefined symbols, reexecute the ar command and order the archive manually by typing in the list of .o files in the appropriate order following the archive name. Be sure to move the .o files with the undefined symbols to the bottom of the archive.

Now, change the directory to ../../bin, and load ENTER by typing:

enter.ld ../src/sfe/sfedit.a <ENTER>

This creates the custom version of ENTER in the current directory (the local tutorial bin directory). When the menu handler executes an ENTER screen, it starts up the custom version in the local directory. If you want, you can install the custom version in the unify/bin directory, and everyone can use it.

To use a different directory or archive name for your version of ENTER, give enter.ld the correct path name to the custom function archive. Use the path name that corresponds to your own custom archive file. If you do not specify the path name to the archive, you see the message:

enter.ld: 1: The customizing archive parameter is missing.

If the archive does not exist, or if it cannot be read, you see:

enter.ld: Unable to read xxxx.

xxxx is the path name of the archive. ENTER is loaded as a single process, and the command file is:

```
if test -z ${UNIFY:?'The Unify environment variable is not set.'};
    then
    exit
fi
if test -r ${1:?'The customizing archive parameter is missing.'};
    then
    ruld TMPLOAD (
         $UNIFY/enter.a (
         $1

    if test -r TMPLOAD; then
         culd ENTER (
              TMPLOAD
         rm TMPLOAD
    fi
else
    echo $Ø: Unable to read $1.
fi
```

The command file loads ENTER in two passes, using the -r flag for the loader. This allows more symbols to load, since many TRS-XENIX loaders have limited symbol table space.

Multiple ENTER Executables

This section describes how to have multiple copies of the ENTER executable, all with different names, and be able to select them from the menu handler. Suppose that you have ten ENTER screens to customize as well as many other standard ENTER screens. Altogether, you have 8ØK of code in your customization routines, and you do not want a single executable approaching 2ØØK on your system.

The solution is to use the standard ENTER with the standard screen forms and divide up the custom screen forms into three or four sets so that the customization code for each set fits into a single executable. Then load different versions of ENTER, one

for each set, and give them all different names. Now you have several smaller executables, each of which is a customized version of ENTER. These custom ENTERs run exactly like the standard ENTER, except that they do not run from a pipe and are executed each time they are called.

To call these customized ENTERs from the menu handler, you must register them. However, the menu handler treats ENTER as a special case. ENTER does not interpret its parameters in exactly the same way as "ordinary" executables do, and the standard ENTER is connected to the menu handler with a pipe. To make the custom ENTERs work properly, you need to know which parameters the menu handler passes to "ordinary" programs and with which parameters ENTER expects to be called. With this information, you can write a shell script that makes the necessary translation and calls the custom ENTER correctly. You can then register the shell script with the menu handler.

The normal method used by the menu handler to call an executable is explained fully in Section 2 but is summarized here. Suppose that you have an ordinary executable name EXEC, containing several programs and using sysrecev. The menu handler would start a program within this executable by passing the following line to the shell:

EXEC T <index> <access level> <language code> <screen name>

The parameters have the following meanings:

Parameter	Usage	Source
T	Not used, present for historical reasons	Constant value
<index>	The prgtab table index for the particular program to start	Executable Maintenance

Parameter	Usage	Source
<access level>	Integer from Ø to 15 that defines whether this user has ADD, INQUIRE, MODIFY, or DELETE privileges	Group Maintenance or Employee Maintenance
<language code>	System language code	System Parameter Maintenance
<screen name>	Name of the screen form associated with this program	Executable Maintenance

ENTER uses all these parameters. The only difference is that the <index> parameter is not used as a prgtab table index. Instead, this parameter is the record number of the target record type for the screen, preceded by a hyphen (-). The record number can be found in the Schema Listing in the "Record List" section of the report, under the column headed "NUMBER." You can also find the record number in the file.h file. All you must do is translate the <index> parameter to the proper record number and call the ENTER executable that has the customization code for that screen form.

For example, suppose that you wanted to retain the standard ENTER for ordinary ENTER screens and have a separate executable for the two custom screen forms discussed earlier in this section. The first step is to rename the custom ENTER, like this:

```
mv ENTER MYENT <ENTER>
```

The next step is to register a new executable with the menu handler that contains two programs -- one for the custom customer

screen form and one for the custom item screen form. After registering this new executable, the Executable Maintenance screen would look something like this:

```
[execmnt]                          UNIFY SYSTEM
                                 5 MAY 1983 - 15:25
                              Executable Maintenance

EXECUTABLE'S NAME: CENTER
USES SYSRECEV ? Y
PROGRAMS:
cmd NAME       HEADING                    SCREEN    DIRECTORY
  Ø cust       Customer Maintenance       customer
  1 itm        Item Maintenance           item
  2
  3
  4
  5
  6
  7
  8
  9


[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE _
```

It is important to indicate y at the USES SYSRECEV ? prompt; otherwise, the parameters will not be passed. Now you should write the CENTER (the name of the executable) shell script that takes the standard parameters from the menu handler and translates the <index> parameter from the prgtab table index number to the target record number for the screen form. Looking at the schema listing, you see that customer is record type 4 and that item is record type 3. Noting on the Executable Maintenance

screen that the index for customer is Ø and that the index for item is 1, you can now write the shell script:

```
case $2 in

Ø) MYENT T -4 $3 $4 $5;;

1) MYENT T -3 $3 $4 $5;;

esac
```

Now, you only need to make the mode of CENTER executable, and you are finished:

```
chmod +x CENTER <ENTER>
```

You can now put the cust and itm programs on any menus or execute them directly from the SELECTION: menu line by typing their names. This same approach can be used for any number of different custom ENTERs.

SQL -- QUERY LANGUAGE

This section explains Unify's major query language interface -- SQL. SQL is a relational inquiry and data manipulation language based on an English keyword syntax. Its structure is easy enough for beginners to use, yet powerful enough for data processing professionals.

All examples in this section are based upon a simple three-table personnel database like any company might use. The tables in this database are:

emp

number	name	dept_no	job	manager	salary	commission

dept

number	name	location

taxes

min_amount	max_amount	base_tax	marginal_rate

The emp table describes employees, giving each employee's number, name, department number, job, manager's employee number, monthly salary, and yearly commission. The dept table lists department numbers, names, and locations. Finally, the taxes table describes a personal income tax schedule, giving the minimum and maximum compensation amounts, base tax, and the marginal tax rate on income above the minimum amount.

SQL Query Facilities

SQL is a powerful English keyword-based language that lets you specify queries of varying complexity. A query consists of phrases (also called "clauses") preceded by keywords. These keywords have special meaning to SQL and so cannot be used for record or field identifiers. See "Keywords" for a list of these reserved keywords. Some phrases are required, and some are optional. Required phrases are:

select	some data (a list of field names)
from	some place (a list of record types)

Optional phrases are:

where	a condition (a true/false statement)
group by	some data (a list of field names)
having	a group condition (a true/false statement)
order by	some data (a list of field names)

The record types and fields for use in selection and calculation are identified by the regular record names and the long field names from the Unify schema design for the database. Long field names are not required to be unique for the entire database, as regular field names must, but only within a specific record type. If you give the same long name to fields in different record types and want to use both fields in the same query, you must qualify each long field name with the name of the record type. This is illustrated in the "General Join" explanation. The complete rules for valid long field names (referred to as "field names" or "fields" in the remainder of this section) are:

. Field names must begin with a letter and contain only letters (both lower-and uppercase are allowed), digits, or underscores (_).

- Field names cannot contain spaces. SQL has no way of distinguishing two separate fields from a single field containing a space.

- Field names must be unique within record type.

SQL lets you type query statements free-form. This means that extra spaces and carriage returns (<ENTER>) may be used freely to improve the readability of a query statement without changing its meaning.

A query is ended with a slash (/), at which time SQL starts evaluating the data you typed. If you enter a valid query, the message:

recognized query!

lets you know that your query is being processed. To exit SQL, type **end** <ENTER>, and you are returned to the menu handler.

If SQL is unable to understand your query, an error message is displayed, giving you an idea of the type of error made. See "Error Messages" for a summary of the error messages and corrective actions. Also see "Editing Query Statements" and "Executing Stored Queries" for instructions on editing, saving, and restarting saved queries.

The following sections take each of the phrases and show you some possible queries. Because of the power of the language, it is not possible to show every kind of query in this manual. For this reason, the formal grammar of SQL is presented later in this section. See the discussion entitled "Reference" for the complete specification of all valid queries.

HELP FEATURES

Unify SQL provides three kinds of help to make learning and using SQL easier. You can get help about the general syntax of SQL, about any specific keyword or keyword phrase, or about the valid record and field names for your database.

To get the first level of help, type **help** <ENTER> at the sql> prompt. The screen displays a list of the valid keywords about which you can get further information.

To get more information about a specific keyword, type **help** followed by the keyword and <ENTER>. Thus, to find out more about the **select** clause, type **help select** <ENTER> at the sql> prompt. The screen shows a description of the **select** clause and its uses.

Information is also available about the record types and fields of your database. To get a list of the valid record names in the current database, type **records** <ENTER> at the sql> prompt. The screen displays a list of the record names you can use with the from clause.

To find the field names for any record type, type **fields** followed by the name of the desired record type and <ENTER>. For example, to list the fields in the emp record type, type **fields emp** <ENTER> at the sql> prompt. The screen shows a list of the field names for this record type.

Following are the results of listing the record and field names for the database you will be studying in the sections to follow, using the SQL help features. Also included is Unify's schema design for the database. Note that the record names come from the regular record names, while the field names come from the long field names.

```
sql> records
emp       dept      taxes
sql> fields emp
NAME                                    TYPE     LENGTH

number                                  INTEGER  4
name                                    STRING   1Ø
dept_no                                 INTEGER  2
job                                     STRING   1Ø
manager                                 INTEGER  4
salary                                  AMOUNT   4
commission                              AMOUNT   4
```

```
sql> fields dept
NAME                                    TYPE     LENGTH

number                                  INTEGER  2
name                                    STRING   15
location                                STRING   15

sql> fields taxes
NAME                                    TYPE     LENGTH

min_amount                              AMOUNT   5
max_amount                              AMOUNT   5
base_tax                                AMOUNT   5
marginal_rate                           FLOAT    42
```

Schema Listing

```
RECORD/FIELD        REF        TYPE     LEN  LONG NAME

emp                 1ØØ                      employee
    *eno                       NUMERIC    4  number
     ename                     STRING    1Ø  name
     edept                     NUMERIC    2  dept_no
     ejob                      STRING    1Ø  job
     emanager                  NUMERIC    4  manager
     esal                      AMOUNT     4  salary
     ecomm                     AMOUNT     4  commission

dept                1Ø                       department
    *dno                       NUMERIC    2  number
     dname                     STRING    15  name
     dloc                      STRING    15  location

taxes               25                       taxes
     minamt                    AMOUNT     5  min_amount
     maxamt                    AMOUNT     5  max_amount
     basetax                   AMOUNT     5  base_tax
     mrgrate                   FLOAT     42  marginal_rate
```

Select From Clause

The simplest kind of SQL query includes a **select** clause and a **from** clause. The select clause lists the fields to be printed, and the from clause tells the record type(s) from which the fields are to come. The fields to be selected must be in the record types in the **from** list.

Example: Select all the fields for each record in the emp record type. This shows the entire contents of the emp record type in the test database. * is an abbreviation for "all the fields."

```
sql> select *
sql> from emp/
recognized query!
```

number	name	dept_no	job	manager	salary	commission
1000	Smith	50	salesman	2100	1500.00	1000.00
2000	Jones	10	clerk	1300	900.00	0.00
1100	Whittaker	20	salesman	2400	2500.00	500.00
2100	Reilly	30	salesman	2400	2500.00	1500.00
1200	O'Neil	20	salesman	1100	1500.00	150.00
2200	Dugan	40	salesman	2400	1650.00	900.00
1300	Schmidt	60	programmer	2400	2500.00	0.00
2300	Klein	20	salesman	1100	1500.00	0.00
1400	Scharf	10	clerk	2000	800.00	0.00
2400	Lee	10	president	2400	7500.00	0.00
1500	Otsaka	60	engineer	1300	1800.00	0.00
2500	Kawasaki	30	salesman	2100	1800.00	1000.00
1600	Dupre	50	clerk	1000	800.00	0.00
2600	Bleriot	10	programmer	1300	1100.00	0.00
1700	Moehr	70	clerk	2400	950.00	0.00
2700	Colucci	40	salesman	2200	2500.00	3000.00
1800	Amato	40	salesman	2200	2000.00	750.00
2800	Fiorella	70	clerk	1700	800.00	0.00
1900	Brown	60	engineer	1300	2000.00	0.00

Example: List the contents of the dept record type.

```
sql> select *
sql> from dept/
recognized query!
```

number	name	location
10	Administration	Dallas
20	Eastern Sales	New York
30	Central Sales	Chicago
40	Western Sales	Los Angeles
50	Marketing	San Francisco
60	Research	Dallas
70	Finance	Dallas

Example: List all the records in the taxes record type.

```
sql> select *
sql> from taxes/
recognized query!
```

min_amount	max_amount	base_tax	marginal_rate
0.00	3399.99	0.00	0.00000
3400.00	4499.99	0.00	0.12000
5500.00	7599.99	252.00	0.14000
7600.00	11599.99	546.00	0.16000
11900.00	15999.99	1234.00	0.19000
16000.00	20199.99	2013.00	0.22000
20200.00	24599.99	2937.00	0.25000
24600.00	29899.99	4037.00	0.29000
29900.00	35199.99	5574.00	0.33000
35200.00	45799.99	7323.00	0.39000
45800.00	59999.99	11457.00	0.44000
60000.00	85599.99	17705.00	0.49000
85600.00	99998.99	30249.00	0.50000

When you use * to select fields, all the fields are returned in system-determined order. When you list the fields explicitly, you can control the sequence and number of fields that are listed.

Example: List the employee number, job, name, and salary for every employee.

```
sql> select number, job, name, salary
sql> from emp/
recognized query!
```

number	job	name	salary
1ØØØ	salesman	Smith	15ØØ.ØØ
2ØØØ	clerk	Jones	9ØØ.ØØ
11ØØ	salesman	Whittaker	25ØØ.ØØ
21ØØ	salesman	Reilly	25ØØ.ØØ
12ØØ	salesman	O'Neil	15ØØ.ØØ
22ØØ	salesman	Dugan	165Ø.ØØ
13ØØ	programmer	Schmidt	25ØØ.ØØ
23ØØ	salesman	Klein	15ØØ.ØØ
14ØØ	clerk	Scharf	8ØØ.ØØ
24ØØ	president	Lee	75ØØ.ØØ
15ØØ	engineer	Otsaka	18ØØ.ØØ
25ØØ	salesman	Kawasaki	18ØØ.ØØ
16ØØ	clerk	Dupre	8ØØ.ØØ
26ØØ	programmer	Bleriot	11ØØ.ØØ
17ØØ	clerk	Moehr	95Ø.ØØ
27ØØ	salesman	Colucci	25ØØ.ØØ
18ØØ	salesman	Amato	2ØØØ.ØØ
28ØØ	clerk	Fiorella	8ØØ.ØØ
19ØØ	engineer	Brown	2ØØØ.ØØ

WHERE CLAUSE

Since you rarely want to list the entire contents of a record type, the **where** clause lets you specify selection

criteria. It compares a field with a constant, expression, or the result of another select clause. These "nested" queries are described in more detail in "Nested Queries." The where clause can also contain a complex boolean expression composed of selection criteria connected by "and" and "or" operators. Precedence can be established by the use of square brackets. The next three sections illustrate the use of the where clause.

Logical Expressions and Operators

Example: List the name and location of department number 7Ø. This illustrates comparison of a field to a numeric constant.

```
sql> select name, location
sql> from dept
sql> where number = 7Ø/
recognized query!

name           |location
-------------------------------
Finance        |Dallas
```

Suppose that you want to find all the departments in Dallas. This requires comparing the location field with the constant string, "Dallas." Unify SQL string constants may contain any combination of the special characters, *, ?, and [], which are used as follows:

? -- The "wild character." The question mark matches any single character. If you wanted to find all the Smiths but did not know if the names were spelled "Smith" or "Smyth," you could type **Sm?th**.

* -- The "wild string." The asterisk matches any string of characters of any length, including zero length strings (also called "null strings").

[...] -- The three dots stand for a set of characters that define a "character class." The character class matches any single character that is a member of the class. Ranges of characters are specified by separating two characters by a hyphen (-). For example, all uppercase letters could be represented by:

[ABCDEFGHIJKLMNOPQRSTUVWXYZ]

or as [A-Z]. All letters, upper- and lowercase together, can be represented as [a-zA-Z]. Other classes can be specified in the same way.

Note that since "Dallas" is shorter than the length of location (which is ten characters long), you must add * to the constant. String constants are also enclosed in single quotation marks (') to distinguish them from field names.

Example: List the name and location of all departments in Dallas.

```
sql> select name, location
sql> from dept
sql> where location = 'Dallas*'/
recognized query!

name           |location
-------------------------------
Administration |Dallas
Research       |Dallas
Finance        |Dallas
```

Example: List the names and jobs of all employees whose names begin with the letters A through M.

```
sql> select name, job
sql> from emp
```

```
sql> where name = '[A-M]*'/
recognized query!

name       |job
-----------------------
Jones      |clerk
Dugan      |salesman
Klein      |salesman
Lee        |president
Kawasaki   |salesman
Dupre      |clerk
Bleriot    |programmer
Moehr      |clerk
Colucci    |salesman
Amato      |salesman
Fiorella   |clerk
Brown      |engineer
```

Example: List the names and jobs of all employees whose names have five or less "nonblank" characters.

```
sql> select name, job
sql> from emp
sql> where name = '?????     '/
recognized query!

name       |job
-----------------------
Smith      |salesman
Jones      |clerk
Dugan      |salesman
Klein      |salesman
Lee        |president
Dupre      |clerk
Moehr      |clerk
Amato      |salesman
Brown      |engineer
```

The where clause can compare two fields with each other, using any of the comparison operators: =, ^=, <, <=, >, and >=.

Example: List the name, job, salary, and commission for employees whose commissions exceed their salaries.

```
sql> select name, job, salary, commission
sql> from emp
sql> where commission > salary/
recognized query!
```

name	job	salary	commission
Colucci	salesman	25ØØ.ØØ	3ØØØ.ØØ

The standard boolean operators, "and" and "or," can connect simple comparisons to form complex expressions. Square brackets indicate which part of the expression should be evaluated first.

Example: List the name, job, salary, and department number for employees who work in department 1Ø and are either clerks or make less than or equal to $12ØØ.ØØ.

```
sql> select name, job, salary, dept_no
sql> from emp
sql> where dept_no = 1Ø and
sql> [job = 'clerk*' or salary <= 12ØØ]/
recognized query!
```

name	job	salary	dept_no
Jones	clerk	9ØØ.ØØ	1Ø
Scharf	clerk	8ØØ.ØØ	1Ø
Bleriot	programmer	11ØØ.ØØ	1Ø

Frequently, you might want to select records based upon a range of values. With the comparison operators, this could be awkward to express. For example, if you wanted to find employees with monthly salaries between $1ØØØ.ØØ and $2ØØØ.ØØ, you would have to use the expression:

```
salary >= 1ØØØ and salary <= 2ØØØ
```

SQL provides a **between** operator to express this more easily.

Example: List the names, salaries, and jobs of employees making between $15ØØ.ØØ and $2ØØØ.ØØ, inclusive.

```
sql> select name, salary, job
sql> from emp
sql> where salary between 15ØØ and 2ØØØ/
recognized query!
```

name	salary	job
Smith	15ØØ.ØØ	salesman
O'Neil	15ØØ.ØØ	salesman
Dugan	165Ø.ØØ	salesman
Klein	15ØØ.ØØ	salesman
Otsaka	18ØØ.ØØ	engineer
Kawasaki	18ØØ.ØØ	salesman
Amato	2ØØØ.ØØ	salesman
Brown	2ØØØ.ØØ	engineer

Not Conditions

Boolean expressions can be negated in whole or in part to select those records that do not match a specified criterion. For instance, suppose that in the research department are engineers, programmers, clerks, typists, and other kinds of employees. To select everyone but engineers would normally require listing each separate job title. The easiest way to do this is to use a not equal comparison operator.

Example: List the department number, name, job, and salary of everyone in department 6Ø who is not an engineer.

```
sql> select dept_no, name, job, salary
sql> from emp
sql> where dept_no = 6Ø and
sql> job ^= 'engineer*' /
recognized query!

dept_no|name      |job       |          salary
-------------------------------------------
     6Ø|Schmidt   |programmer|         25ØØ.ØØ
```

Example: List the name, job, and salary of each employee who is not a salesman or who makes less than $2ØØØ.ØØ. This uses the **not** operator to negate an entire expression. The square brackets indicate that the operator applies to the whole expression.

```
sql> select name, job, salary
sql> from emp
sql> where not[job = 'salesman*' or salary >= 2ØØØ]/
recognized query!

name      |job       |          salary
-----------------------------------------
Jones     |clerk     |           9ØØ.ØØ
Scharf    |clerk     |           8ØØ.ØØ
Otsaka    |engineer  |          18ØØ.ØØ
Dupre     |clerk     |           8ØØ.ØØ
Bleriot   |programmer|          11ØØ.ØØ
Moehr     |clerk     |           95Ø.ØØ
Fiorella  |clerk     |           8ØØ.ØØ
```

Set Inclusion

In many queries, you may want to compare a field to a list of values, not just a single value. For example, consider selecting all the employees who are in departments 2Ø, 3Ø, or 4Ø. With the standard operators, this becomes a sequence of equalities connected by **ors**, such as:

dept_no = 2Ø or dept_no = 3Ø or dept_no = 4Ø ...

As with ranges, this could become awkward. SQL provides a set **inclusion notation** to make this kind of query easier.

Example: List the name, job, and department number for any employee in departments 2Ø, 3Ø, or 4Ø.

```
sql> select name, job, dept_no
sql> from emp
sql> where dept_no in <2Ø, 3Ø, 4Ø>/
recognized query!
```

name	job	dept_no
Whittaker	salesman	2Ø
Reilly	salesman	3Ø
O'Neil	salesman	2Ø
Dugan	salesman	4Ø
Klein	salesman	2Ø
Kawasaki	salesman	3Ø
Colucci	salesman	4Ø
Amato	salesman	4Ø

The set inclusion notation can also be used to group and conditions together. This is useful if you want to select based on a combination of fields -- for example, all clerks in department 1Ø and all programmers in department 6Ø. Using standard notation, this would be expressed as:

```
[job = 'clerk*' and dept_no = 1Ø] or
[job = 'programmer*' and dept_no = 6Ø]
```

The set inclusion operator can express this kind of statement more easily.

Example: List the name, job, salary, and department number of each clerk in department 1Ø and each programmer in

department 6Ø. Note the use of parentheses to group the set of constants.

```
sql> select name, job, salary, dept_no
sql> from emp
sql> where <job, dept_no> is in ( <'clerk*', 1Ø>,
sql> <'programmer*', 6Ø>)/
recognized query!
```

name	job	salary	dept_no
Jones	clerk	9ØØ.ØØ	1Ø
Schmidt	programmer	25ØØ.ØØ	6Ø
Scharf	clerk	8ØØ.ØØ	1Ø

Sets of constants like:

<'clerk*', 1Ø>

are called "literal tuples" in relational terminology. A literal tuple is simply a group of constants that describe values for specific fields in a record. The fields must be identified by preceding the tuples with a field specification list, as in the above example.

UNIQUE OPERATOR

If a query does not select a primary key field from one of the record types, the query might produce rows that are exact duplicates of each other, since only the primary key must be unique. For example, there are many clerks, programmers, etc. in the emp file, so a query that selected the job field would produce duplicate rows. Sometimes, these duplicates are not desired. The **unique** operator lets you suppress duplicate rows in a query result.

Example: List the different job titles in the company. Note that a side effect of the unique operator is to sort the result of the query.

```
sql> select unique job
sql> from emp/
recognized query!

job
-----------
clerk
engineer
president
programmer
salesman
```

Example: List the department number and job classification of employees in departments 1Ø, 2Ø, and 3Ø or employees who make more than $2ØØØ.ØØ. Note that this query produces duplicate rows.

```
sql> select dept_no, job
sql> from emp
sql> where dept_no in <1Ø, 2Ø, 3Ø> or
sql> salary > 2ØØØ/
recognized query!

dept_no|job
-------------------
     1Ø|clerk
     2Ø|salesman
     3Ø|salesman
     2Ø|salesman
     6Ø|programmer
     2Ø|salesman
     1Ø|clerk
     1Ø|president
     3Ø|salesman
     1Ø|programmer
     4Ø|salesman
```

Example: Perform the preceding query again, this time eliminating the duplicates. As before, the rows are sorted.

```
sql> select unique dept_no, job
sql> from emp
sql> where dept_no in <1Ø, 2Ø, 3Ø> or
sql> salary > 2ØØØ/
recognized query!

dept_no|job
-------------------
     1Ø|clerk
     1Ø|president
     1Ø|programmer
     2Ø|salesman
     3Ø|salesman
     4Ø|salesman
     6Ø|programmer
```

ARITHMETIC EXPRESSIONS

Arithmetic expressions let you calculate values using the operators: +, -, *, and /. Both constants and fields may be used in arithmetic expressions, which are allowed on fields of all types except STRING and COMB. Arithmetic expressions may be used wherever a simple field is allowed in select, where, and having clauses.

Example: List the name, job, and total of salary plus commission for employees in departments 2Ø, 3Ø, and 4Ø (the sales departments).

```
sql> select name, job, salary + commission
sql> from emp
sql> where dept_no in <2Ø, 3Ø, 4Ø>/
recognized query!
```

name	job	salary+commission
Whittaker	salesman	3ØØØ.ØØ
Reilly	salesman	4ØØØ.ØØ
O'Neil	salesman	165Ø.ØØ
Dugan	salesman	255Ø.ØØ
Klein	salesman	15ØØ.ØØ
Kawasaki	salesman	28ØØ.ØØ
Colucci	salesman	55ØØ.ØØ
Amato	salesman	275Ø.ØØ

Example: Find those salesmen who have not earned commissions of at least $1ØØ.ØØ plus half their monthly salaries. List each name, salary, commission they actually earned, and what they should have earned.

```
sql> select name, salary, commission, (salary * .5) + 1ØØ
sql> from emp
sql> where commission < (salary * .5) + 1ØØ and
sql> job = 'salesman*'/
recognized query!
```

name	salary	commission	(salary*.5)+1ØØ
Whittaker	25ØØ.ØØ	5ØØ.ØØ	135Ø.ØØ
O'Neil	15ØØ.ØØ	15Ø.ØØ	85Ø.ØØ
Dugan	165Ø.ØØ	9ØØ.ØØ	925.ØØ
Klein	15ØØ.ØØ	Ø.ØØ	85Ø.ØØ
Amato	2ØØØ.ØØ	75Ø.ØØ	11ØØ.ØØ

Example: Suppose that you make $3ØØØ.ØØ per month and earn commissions of $5ØØØ.ØØ per year. Compute your total compensation, base tax amount, and marginal tax amount, using the tax tables in the taxes file.

```
sql> select (3ØØØ*12)+5ØØØ, base_tax,
sql> (((3ØØØ*12)+5ØØØ)-min_amount)*marginal_rate
```

```
sql> from taxes
sql> where (3ØØØ*12)+5ØØØ between min_amount and max_amount/
recognized query!
(3ØØØ*12)+5ØØØ|base_tax|(((3ØØØ*12)+5ØØØ)-min_amount)*marginal_rate
--------------------------------------------------------------------
   41ØØØ.ØØØØØ| 7323.ØØ|                                    2262.ØØ
```

ORDER BY CLAUSE

All the previous queries displayed their results in an order determined by SQL. Even though the unique operator sorted its output, you still had no control over the order of output. The **order by** clause lets you specify the exact sequence of the rows that result from a query. The default sort sequence is ascending, with STRING fields sorted in alphabetic order from A to Z.

Example: List every employee's number, name, and job, sorted by employee number.

```
sql> select number, name, job
sql> from emp
sql> order by number/
recognized query!
```

number	name	job
1000	Smith	salesman
1100	Whittaker	salesman
1200	O'Neil	salesman
1300	Schmidt	programmer
1400	Scharf	clerk
1500	Otsaka	engineer
1600	Dupre	clerk
1700	Moehr	clerk
1800	Amato	salesman
1900	Brown	engineer
2000	Jones	clerk
2100	Reilly	salesman
2200	Dugan	salesman
2300	Klein	salesman
2400	Lee	president
2500	Kawasaki	salesman
2600	Bleriot	programmer
2700	Colucci	salesman
2800	Fiorella	clerk

You can sort by more than one field and specify the direction, whether ascending or descending, for each field in the sort.

Example: List every employee's department number, name, and job, by ascending name within descending department number.

```
sql> select dept_no, name, job
sql> from emp
sql> order by dept_no desc, name asc/
recognized query!
```

dept_no	name	job
7Ø	Fiorella	clerk
7Ø	Moehr	clerk
6Ø	Brown	engineer
6Ø	Otsaka	engineer
6Ø	Schmidt	programmer
5Ø	Dupre	clerk
5Ø	Smith	salesman
4Ø	Amato	salesman
4Ø	Colucci	salesman
4Ø	Dugan	salesman
3Ø	Kawasaki	salesman
3Ø	Reilly	salesman
2Ø	Klein	salesman
2Ø	O'Neil	salesman
2Ø	Whittaker	salesman
1Ø	Bleriot	programmer
1Ø	Jones	clerk
1Ø	Lee	president
1Ø	Scharf	clerk

AGGREGATE FUNCTIONS

SQL provides five different built-in aggregate functions to allow calculation of aggregate items in a query result. The functions are: **count(*)**, **min**, **max**, **sum**, and **avg**. Aggregate functions are only valid in select or having clauses -- you may not use an aggregate function directly in a where clause, although you can usually achieve the same effect by using a nested query. This is described in more detail in "Nested Queries."

An aggregate function only applies to a group of records with a common characteristic -- for example, the average salary of employees in department 1Ø. In this case, the common characteristic is the department number. When using an aggregate function in the select clause, be careful to select only fields that remain constant for each member of the group. It means

nothing to select employee name along with the average salary for a department, since each employee's name is different. To list the names of employees who make the average salary, you must use a nested query. Refer to "Nested Queries" for further information.

Aggregate functions are most commonly used in conjunction with the **group by** clause. A group by clause lets you partition the selected records into groups for which the indicated aggregate functions are computed. If there is no explicit group by clause, any aggregate functions are assumed to apply to the entire set of selected records.

Example: Compute and list the total number of employees in department 1Ø.

```
sql> select count (*)
sql> from emp
sql> where dept_no = 1Ø/
recognized query!

 count(*)
----------
         4
```

Example: Compute and list the minimum and maximum salaries throughout the entire company.

```
sql> select min(salary), max(salary)
sql> from emp/
recognized query!

     min(salary)|   max(salary)
--------------------------------
          8ØØ.ØØ|       75ØØ.ØØ
```

Example: List the job and average of salary plus commission for all salesmen. The job can be listed in this example, because all the selected records have the same job -- salesman.

```
sql> select job, avg (salary + commission)
sql> from emp
sql> where job = 'salesman*'/
recognized query!

job       |avg(salary+commission)
----------------------------------
salesman  |               2916.67
```

Example: Figure the amount that engineers and programmers are paid on a yearly basis. This requires selecting all programmers and engineers, multiplying their monthly salaries by 12, adding in their commissions (presumably Ø), and adding up the total.

```
sql> select sum((salary*12)+commission)
sql> from emp
sql> where job = 'engineer*' or job = 'programmer*'/
recognized query!

sum((salary*12)+commission)
---------------------------
                  888ØØ.ØØ
```

GROUP BY CLAUSE

The **group by** clause allows computation of aggregate functions on groups of records that have common characteristics. Using a group by clause without an aggregate function has no meaning. The effect of a group by clause is to sort the selected rows by the indicated fields and then perform the aggregate functions at each level break. This results in sorted output.

Example: For each department, list the department number, count of employees, and sum of salary plus commission.

```
sql> select dept_no, count(*), sum(salary+commission)
sql> from emp
sql> group by dept_no/
recognized query!
```

dept_no	count(*)	sum(salary+commission)
10	4	10300.00
20	3	6150.00
30	2	6800.00
40	3	10800.00
50	2	3300.00
60	3	6300.00
70	2	1750.00

More than one field can be used in a group by clause, resulting in more divisions in the calculation.

Example: List the department number, job, count of employees, and average salary for each job title within department, for departments 10, 20, and 30.

```
sql> select dept_no, job, count(*), avg(salary)
sql> from emp
sql> where dept_no is in <10, 20, 30>
sql> group by dept_no, job/
recognized query!
```

dept_no	job	count(*)	avg(salary)
10	clerk	2	850.00
10	president	1	7500.00
10	programmer	1	1100.00
20	salesman	3	1833.33
30	salesman	2	2150.00

Example: Compute the average yearly amount paid to salesmen in each department. The yearly amount is given by the monthly salary times 12, plus the commission. Also list the department number and count of salesmen in each department.

```
sql> select dept_no, count(*), avg((salary*12)+commission)
sql> from emp
sql> where job = 'salesman*'
sql> group by dept_no/
recognized query!

dept_no| count(*)|avg((salary*12)+commission)
-------------------------------------------
     2Ø|        3|                  22216.67
     3Ø|        2|                  27Ø5Ø.ØØ
     4Ø|        3|                  2615Ø.ØØ
     5Ø|        1|                  19ØØØ.ØØ
```

Aggregate functions can also be applied to the result of other aggregate functions. This lets you compute such items as the "maximum average" or the "average count." When used in this way, aggregate functions require a group by clause. The result is computed as follows. First, all qualifying records are selected using the where clause, if any. Then, they are sorted according to the fields in the group by clause, and the inner aggregate function is computed. The outer function is then applied to these results. Since this second level of computation removes all identity from the groups, a nested query is required to list fields other than the function result.

Example: List the average department size in the company.

```
sql> select avg(count(*))
sql> from emp
sql> group by dept_no/
recognized query!
```

```
  avg(count(*))
----------------
        2.71429
```

Example: List the maximum average monthly salary for all jobs except "president."

```
sql> select max(avg(salary))
sql> from emp
sql> where job ^= 'president*'
sql> group by job/
recognized query!

max(avg(salary))
-----------------
         1938.89
```

NESTED QUERIES

Nested queries let you ask questions that cannot be answered using the SQL capabilities discussed so far. Nesting lets you use the results of one query as input in another so that you can use the results of one question in answering another. Suppose, for example, that you wanted to find the name of the employee with the largest salary. Using the capabilities presented so far, you could perform a query to find the maximum salary as follows:

```
select max(salary+commission) from emp/
```

You could also find the employees who make a constant salary, such as all those who earn $25ØØ.ØØ:

```
select name from emp
where salary+commission = 25ØØ/
```

The only way to find the employee who makes the maximum is to do the first query, then use the resulting value of salary

plus commission as the constant value in the second query. Nesting lets you do this automatically.

Example: Find the name and job of the employee who makes the maximum salary plus commission.

```
sql> select name, job
sql> from emp
sql> where salary+commission =
sql> select max(salary+commission)
sql> from emp/
recognized query!

name       |job
-----------------------
Lee        |president
```

This query works by first evaluating the inner query to get a value for the maximum salary plus commission. This value is then used as a constant in the outer query, which finds the employees who make that amount.

Nested queries can also use the **set inclusion** facilities described in "Set Inclusion" to shorten the query statement. For example, suppose that someone not in department 1Ø, the company headquarters, wonders if department 1Ø's employee make more than employees with the same jobs in other departments. A nested query using set inclusion can be used to see if this is true.

Example: List the name, job, and salary of employees not in department 1Ø who have the same job and salary as any employee in department 1Ø.

```
sql> select name, job, salary
sql> from emp
sql> where dept_no ^= 1Ø
sql> and <job, salary> is in
```

```
sql> select job, salary
sql> from emp
sql> where dept_no = 1Ø/
recognized query!

name      |job       |          salary
-----------------------------------------
Dupre     |clerk     |          8ØØ.ØØ
Fiorella  |clerk     |          8ØØ.ØØ
```

Queries can be nested to any level, not just the two illustrated above. The following four-level query finds the employee with the second highest compensation. The method used finds the employee with the maximum compensation and then finds the maximum compensation among those employees left. This also shows that you can compare an expression with the results of a nested query. In trying to understand nested queries, you should start with the innermost query and work your way out, rather than trying to work from the outside in.

Example: List the department number, name, job, and salary plus commission of the employee with the second highest compensation.

```
sql> select dept_no, name, job, salary+commission
sql> from emp
sql> where salary+commission =
sql> select max(salary+commission)
sql> from emp
sql> where name ^=
sql> select name
sql> from emp
sql> where salary+commission =
sql> select max(salary+commission)
sql> from emp/
recognized query!

dept_no|name      |job       |salary+commission
-----------------------------------------------
     4Ø|Colucci   |salesman  |          55ØØ.ØØ
```

Nested queries can also be used as part of more complex expressions. In this case, the inner query must end with a semicolon (;) so that SQL can determine where the nested query ends and the rest of the boolean expression begins. The following query compares the top salesman in the company with all salesmen in department 2Ø.

Example: List the department number, name, job, total compensation, and commission for the salesman with the maximum total compensation and for all the salesmen in department 2Ø. Sort the result by salary within commission in descending sequence, so that high earners come first.

```
sql> select dept_no, name, job, salary+commission, commission
sql> from emp
sql> where salary+commission =
sql> select max(salary+commission)
sql> from emp
sql> where job = 'salesman*';
sql> or [job = 'salesman*' and dept_no = 2Ø]
sql> order by commission desc, salary desc/
recognized query!
```

dept_no	name	job	salary+commission	commission
4Ø	Colucci	salesman	55ØØ.ØØ	3ØØØ.ØØ
2Ø	Whittaker	salesman	3ØØØ.ØØ	5ØØ.ØØ
2Ø	O'Neil	salesman	165Ø.ØØ	15Ø.ØØ
2Ø	Klein	salesman	15ØØ.ØØ	Ø.ØØ

HAVING CLAUSE

The **having** clause lets you select some groups formed by a previous group by clause and reject others, based upon the results of another selection using one or more of the aggregate

functions. This capability is the same as being able to use an aggregate function in a where clause. For example, you can use the having clause to select departments in which the average salary is over $2ØØØ.ØØ.

Example: List the department number and average salary for departments with average salaries over $2ØØØ.ØØ.

```
sql> select dept_no, avg(salary)
sql> from emp
sql> group by dept_no
sql> having avg(salary) > 2ØØØ/
recognized query!

dept_no|     avg(salary)
-------------------------
     1Ø|          2575.ØØ
     3Ø|          215Ø.ØØ
     4Ø|          2Ø5Ø.ØØ
     6Ø|          21ØØ.ØØ
```

When a query contains both having and where clauses, the query is evaluated as follows: first, the where clause is applied to select qualifying records; next, the groups indicated by the group by clause are formed; then the having clause is applied to select qualifying groups.

Example: List the department number and number of salesmen for each department with more than two salesmen.

```
sql> select dept_no, count(*)
sql> from emp
sql> where job = 'salesman*'
sql> group by dept_no
sql> having count(*) > 2/
recognized query!
```

```
dept_no| count(*)
-------------------
     2Ø|         3
     4Ø|         3
```

In "Arithmetic Expressions" you selected individual salesmen whose commissions were below par. Using a having clause that compares the result of two aggregate functions for each department, you can find the departments where commissions are at or above par across the whole department.

Example: List the department number, average salary, average commission, and target commission level (half the salary plus $1ØØ.ØØ) for departments whose average commission is at or above the target level.

```
sql> select dept_no,avg(salary),avg(commission),
sql> avg((salary*Ø.5)+1ØØ)
sql> from emp
sql> where job = 'salesman*'
sql> group by dept_no
sql> having avg(commission) >= avg((salary*Ø.5)+1ØØ)/
recognized query!

dept_no|    avg(salary)|avg(commission)|avg((salary*Ø.5)+1ØØ)
-------------------------------------------------------------
     3Ø|        215Ø.ØØ|        125Ø.ØØ|              1175.ØØ
     4Ø|        2Ø5Ø.ØØ|        155Ø.ØØ|              1125.ØØ
     5Ø|        15ØØ.ØØ|        1ØØØ.ØØ|               85Ø.ØØ
```

The having clause can also contain nested queries. A query nested in a having clause is evaluated in the same way as a query nested in a where clause. This kind of query lets you answer questions like, "Which departments have an average salary less than the average for all departments?"

Example: List the department number and average salary for departments with average salaries of less than the average for all departments.

```
sql> select dept_no, avg(salary)
sql> from emp
sql> group by dept_no
sql> having avg(salary) <
sql> select avg(salary)
sql> from emp/
recognized query!

dept_no|    avg(salary)
-------------------------
     2Ø|         1833.33
     5Ø|         115Ø.ØØ
     7Ø|          875.ØØ
```

Nested queries can be used in both the where and having clauses at the same time. The following query finds the employees that work in the department with the highest average salary. This is performed in three steps -- first, finding the maximum average salary for all departments; second, finding the department with that average salary; and finally, finding all the employees in that department.

Example: List the name, job, and salary for employees in the department with the highest average salary.

```
sql> select name, job, salary
sql> from emp
sql> where dept_no =
sql> select dept_no
sql> from emp
sql> group by dept_no
sql> having avg(salary) =
sql> select max(avg(salary))
sql> from emp
sql> group by dept_no/
recognized query!
```

name	job	salary
Jones	clerk	9ØØ.ØØ
Scharf	clerk	8ØØ.ØØ
Lee	president	75ØØ.ØØ
Bleriot	programmer	11ØØ.ØØ

JOIN QUERIES

Until now, all the queries discussed have involved only a single record type. However, a SQL statement may list fields from any number of record types in a single query. Queries that list fields from several record types are called "join" queries, because they combine different record types. The record types involved in the query are listed in the from clause, in any convenient order. SQL then determines the most efficient method of performing the selection and qualification. The following two sections illustrate various methods of joining record types.

General Join

The fundamental concept underlying join queries is that of the "Cartesian product." Conceptually, a join query first forms the Cartesian product of the record types and then "filters" the results with conditions in the where clause. Thus, a join query without a where clause lists the actual Cartesian product of the record types.

The Cartesian product is best illustrated by an example. Consider three records, a, b, and c. Record type c is the Cartesian product of a and b. It is formed by taking every possible combination of the records in record types a and b.

Type a	Type b	Type c
aaa	ccc	aaa ccc
bbb	ddd	aaa ddd
	eee	aaa eee
		bbb ccc
		bbb ddd
		bbb eee

In this case, record type a has two records, record type b has three records, and the Cartesian product, record type c, has six records. If many records existed, performing a join by actually forming the Cartesian product and then selecting would take an unreasonable amount of time. For example, suppose that you were joing two record types, one with 5ØØØ records and another with 5Ø,ØØØ records. The Cartesian product of these two record types is 25Ø,ØØØ,ØØØ records. This would take days to process.

Unify SQL, however, can perform join selections on files of this size in seconds. To achieve this speed, each query is optimized to take advantage of the available UNITRIEVE access methods, including hashing, explicit relationships, and B-trees. These multiple access methods give you the flexibility to "tune" the performance of your database by adding or changing access methods so that in practice, forming the Cartesian product is not required.

The first query example in this section is simply an illustration of a Cartesian product with no selection. Notice that if a field name is not unique, it must be preceded by the name of the record type. To list the employee name, you must use emp.name, since the dept record type also has a field called name.

Example: List the employee name and department location for the Cartesian product of the employee and department record types. (The entire result is not listed in the sample below.)

```
sql> select emp.name, location
sql> from emp, dept/
recognized query!

emp.name   |location
---------------------------
Smith      |Dallas
Smith      |New York
Smith      |Chicago
Smith      |Los Angeles
Smith      |San Francisco
Smith      |Dallas
Smith      |Dallas
Jones      |Dallas
Jones      |New York
Jones      |Chicago
Jones      |Los Angeles
Jones      |San Francisco
Jones      |Dallas
Jones      |Dallas
Whittaker  |Dallas
Whittaker  |New York
Whittaker  |Chicago
Whittaker  |Los Angeles
Whittaker  |San Francisco
Whittaker  |Dallas
Whittaker  |Dallas
.
.
.
```

Note that the first employee is listed with the location of every department, and so on. Of course, this is not very useful information. The real power of a relational database management system is in its ability to qualify the results of a join so that you can list only the department information for the department in which the employee works. This is done by adding a where clause that selects only those rows of the Cartesian product where the department number in the emp record type is the same as the department number in the dept record type.

There are two things to notice about this query. First, the fields **name** and **number** must be preceded with record type names, since both emp and dept have fields with these names. The dept_no field does not require qualification, since it only occurs in the emp record type. Second, the expression dept.* selects all the fields in the dept record type without having to list each one explicitly.

Example: List the employee name and all the fields from the department in which the employee works.

```
sql> select emp.name, dept.*
sql> from emp, dept
sql> where dept_no = dept.number/
recognized query!
```

emp.name	number	name	location
Smith	5Ø	Marketing	San Francisco
Jones	1Ø	Administration	Dallas
Whittaker	2Ø	Eastern Sales	New York
Reilly	3Ø	Central Sales	Chicago
O'Neil	2Ø	Eastern Sales	New York
Dugan	4Ø	Western Sales	Los Angeles
Schmidt	6Ø	Research	Dallas
Klein	2Ø	Eastern Sales	New York
Scharf	1Ø	Administration	Dallas
Lee	1Ø	Administration	Dallas
Otsaka	6Ø	Research	Dallas
Kawasaki	3Ø	Central Sales	Chicago
Dupre	5Ø	Marketing	San Francisco
Bleriot	1Ø	Administration	Dallas
Moehr	7Ø	Finance	Dallas
Colucci	4Ø	Western Sales	Los Angeles
Amato	4Ø	Western Sales	Los Angeles
Fiorella	7Ø	Finance	Dallas
Brown	6Ø	Research	Dallas

The where clause in a join query can contain expressions that use fields from any of the record types involved in the

join. The expression is not limited to being an equality, as the following query shows.

Example: List the name, yearly salary, and bracketing amounts from the taxes record type between which the yearly salary falls.

```
sql> select name, salary*12, min_amount, max_amount
sql> from emp, taxes
sql> where salary*12 between min_amount and max_amount/
recognized query!
```

name	salary*12	min_amount	max_amount
Smith	18000.00	16000.00	20199.99
Jones	10800.00	7600.00	11599.99
Whittaker	30000.00	29900.00	35199.99
Reilly	30000.00	29900.00	35199.99
O'Neil	18000.00	16000.00	20199.99
Dugan	19800.00	16000.00	20199.99
Schmidt	30000.00	29900.00	35199.99
Klein	18000.00	16000.00	20199.99
Scharf	9600.00	7600.00	11599.99
Lee	90000.00	85600.00	99998.99
Otsaka	21600.00	20200.00	24599.99
Kawasaki	21600.00	20200.00	24599.99
Dupre	9600.00	7600.00	11599.99
Bleriot	13200.00	11900.00	15999.99
Moehr	11400.00	7600.00	11599.99
Colucci	30000.00	29900.00	35199.99
Amato	24000.00	20200.00	24599.99
Fiorella	9600.00	7600.00	11599.99
Brown	24000.00	20200.00	24599.99

Now that you have seen how to find the proper tax bracket, you can easily compute the tax liability for each employee. There is no fundamental limit on the number of tables that can be joined at the same time, so you can list the employee's name, location, total yearly compensation, and tax amount in the same

query. The expressions that calculates the tax amount is somewhat long, so the column heading has been edited to make the result fit on one page.

Example: List each employee's name, location, total yearly compensation, and tax bill.

```
sql> select emp.name, location, (salary*12)+commission,
sql> base_tax+(((salary*12)+commission)-min_amount)
sql> *marginal_rate
sql> from emp, taxes, dept
sql> where (salary*12)+commission between min_amount and
sql> max_amount and dept_no = dept.number/
recognized query!
```

emp.name	location	(salary*12)+commission	tax_amount
Smith	San Francisco	19ØØØ.ØØ	2673.ØØ
Jones	Dallas	1Ø8ØØ.ØØ	1Ø58.ØØ
Whittaker	New York	3Ø5ØØ.ØØ	5772.ØØ
Reilly	Chicago	315ØØ.ØØ	61Ø2.ØØ
O'Neil	New York	1815Ø.ØØ	2486.ØØ
Dugan	Los Angeles	2Ø7ØØ.ØØ	3Ø62.ØØ
Schmidt	Dallas	3ØØØØ.ØØ	56Ø7.ØØ
Klein	New York	18ØØØ.ØØ	2453.ØØ
Scharf	Dallas	96ØØ.ØØ	866.ØØ
Lee	Dallas	9ØØØØ.ØØ	32449.ØØ
Otsaka	Dallas	216ØØ.ØØ	3287.ØØ
Kawasaki	Chicago	226ØØ.ØØ	3537.ØØ
Dupre	San Francisco	96ØØ.ØØ	866.ØØ
Bleriot	Dallas	132ØØ.ØØ	1481.ØØ
Moehr	Dallas	114ØØ.ØØ	1154.ØØ
Colucci	Los Angeles	33ØØØ.ØØ	6597.ØØ
Amato	Los Angeles	2475Ø.ØØ	4Ø8Ø.5Ø
Fiorella	Dallas	96ØØ.ØØ	866.ØØ
Brown	Dallas	24ØØØ.ØØ	3887.ØØ

Self Join

Sometimes a record type must be joined to itself. In the

sample database, the emp record type contains a field that indicates the employee's manager. This is simply the number of another employee. For example, employee number 21ØØ, Reilly, is the manager of Smith and Kawasaki. You could therefore join the emp record type to itself, using the employee's number and manager's number.

The best way to think of queries like this is to imagine that there are two copies of the emp record type -- one that contains employees, and one that contains managers. SQL does not actually make a copy but achieves the same effect by letting you give a record type a temporary name. You are free to join these record types just like "real" ones. The following query uses the record types emp and mgr, a temporary name given to the emp record type.

Example: List employees' names and salaries and their managers' names and salaries for employees who make more than their managers.

```
sql> select emp.name, emp.salary, mgr.name, mgr.salary
sql> from emp, emp mgr
sql> where emp.salary > mgr.salary and
sql> emp.manager = mgr.number/
recognized query!
```

emp.name	emp.salary	mgr.name	mgr.salary
Colucci	25ØØ.ØØ	Dugan	165Ø.ØØ
Amato	2ØØØ.ØØ	Dugan	165Ø.ØØ

Self joins can also be used between many different record types, not just two. Suppose you discovered that when managers are in the same location as their employees, employee relations are improved, and productivity increases 5Ø%. You would naturally want to discover which employees are in different locations than their managers. Conceptually, this query requires joining four tables -- one for employees, one for the departments

in which they are located, one for managers, and one for departments in which managers are located.

Example: List the employees' names and locations and their managers' names and locations for those employees in a different location than their managers.

```
sql> select emp.name, dept.location, mgr.name,
sql> mgr_dept.location
sql> from emp, dept, emp mgr, dept mgr_dept
sql> where emp.manager = mgr.number
sql> and emp.dept_no = dept.number
sql> and mgr.dept_no = mgr_dept.number
sql> and dept.location ^= mgr_dept.location/
recognized query!
```

emp.name	dept.location	mgr.name	mgr_dept.location
Smith	San Francisco	Reilly	Chicago
Whittaker	New York	Lee	Dallas
Reilly	Chicago	Lee	Dallas
Dugan	Los Angeles	Lee	Dallas

VARIABLE QUERIES

Regular nested queries work "from the inside out," evaluating the innermost query and passing the resulting value(s) out to the next outer query. Each query is performed only once. **Variable queries** reverse this procedure, working "from the outside in." This lets you perform an inner query multiple times using values from the outer query. Suppose you were doing a comparative study of salary and commission levels, and you wanted to know which employees made more than 25% of the total paid to everyone else with the same job. To make sure that the results were significant, you would choose only the job categories having at least four employees. You would have to perform the following steps to determine this.

1. Find the job categories where there are at least four employees.

2. For each employee holding a job in these categories:

   a. Compute the total of salary plus commission of everyone else with the same job. Leave out the salary and commission of the current employee.

   b. If the current employee earns more than Ø.25 times this total, select this employee.

The query to answer this question works in just this way. The outer query chooses the current employee, based upon whether the employee belongs to a job category with at least three others in it and executes the inner query using that employee's job and number to exclude the employee from the calculation of total salary plus commission for that job category. The total is multiplied by Ø.25, and if the current employee's salary exceeds that number, the employee is chosen.

These kinds of queries are called "variable queries" because the inner query varies, depending upon a record type in the outer query. In this example, you are joining the emp record type to itself and must give the emp in the outer query a temporary name so that it is apparent in the inner query which emp record you are referencing.

Example: Find the job categories which include at least four employees and then list the name, job, and salary (including commission) for each employee in the group whose total salary and commission is more than 25% of the total of the remaining employees' salary and commission.

```
sql> select name, job, salary+commission
sql> from emp x
sql> where job is in select job
sql> from emp
sql> group by job
sql> having count(*) >= 4;
sql> and salary+commission > select
```

```
sql> sum(salary+commission) * Ø.25
sql> from emp
sql> where emp.job = x.job and
sql> emp.number ^= x.number
sql> group by job/
recognized query!
```

name	job	salary+commission
Jones	clerk	9ØØ.ØØ
Moehr	clerk	95Ø.ØØ
Colucci	salesman	55ØØ.ØØ

Unify SQL Extensions

Unify SQL provides several extensions that adapt it for interactive query development in the TRS-XENIX environment. Probably the most important extension for interactive querying of a database is the ability to edit the last query, either to correct minor errors or to try a slight variation of the last query. "Editing Query Statements" describes the Unify SQL editor interface that lets you switch back and forth easily between the editor of your choice and SQL.

"Executing Stored Queries" describes the mechanism for executing stored queries. Saving a query in a text file for later execution lets you perform often-repeated queries without having to retype them. Access to the powerful TRS-XENIX command interpreter, the shell, is provided by an "escape" capability described in "Executing TRS-XENIX Commands." Finally, because of the many text processing programs available, it is sometimes desirable to save data into an ASCII text file and load it back into the database. Saving data is discussed in "Saving Data to TRS-XENIX Files," and reloading data into the database is discussed in "Loading Data from TRS-XENIX Files."

EDITING QUERY STATEMENTS

SQL stores the last query statement entered in a buffer so that you can edit it to correct mistakes, save it in a file for later execution, or read in an entirely new query. To edit the current query buffer, type **edit** <ENTER> at the sql> prompt.

If the buffer is empty, you are given an empty file to use. The file is created in the temporary work area (/tmp) with a unique name based on the process id so as not to conflict with other users.

The editor vi is used by default, but you can change this with the EDIT environment variable. See the Introduction in this Reference Manual for information about this and other Unify parameters. When in the editor, you can modify the current query,

enter a new query, read in a completely different query from a text file, or write the query to a text file. When you exit the editor, you can execute the query in the buffer by typing **restart** <ENTER> at the sql> prompt. For information on the vi editor, refer to Appendix B in this manual or the TRS-XENIX System Fundamentals Manual.

EXECUTING STORED QUERIES

In any application, there is a standard set of queries used all the time. The ability to store queries and execute them later is very important.

You can create a query interactively using SQL and then use edit to save it, or you can use an editor outside of SQL to create the query. In either case, the result is a text file that contains the query. SQL provides two ways of executing such queries. Suppose that your stored query, which computes the maximum salary for all employees, is in a file called maxsal.

From the TRS-XENIX shell, you can type:

SQL maxsal <ENTER>

and the query in maxsal is executed. The results and any error messages are sent to the standard output (the terminal from which SQL was executed). These can, of course, be piped or redirected using standard TRS-XENIX shell syntax.

The second method is from within SQL, where you can start a stored query by typing **start** followed by the name of the file containing the query and <ENTER>. The entry for the above example would be:

sql> **start maxsal** <ENTER>

The query runs just as if you had typed it from the terminal, except that it is not put into the edit buffer. Thus, an edit after a start lets you edit the query (if any) typed before the start command was issued.

Note that any valid SQL command may be placed in a stored query, including TRS-XENIX shell commands and **start**, which lets you run other stored queries.

EXECUTING TRS-XENIX COMMANDS

TRS-XENIX commands can be directed to the shell by simply preceding them with an exclamation point (!). For example, to cat the maxsal file from within SQL, type:

!cat maxsal <ENTER>

at the sql> prompt. The TRS-XENIX command is executed, and the sql> prompt is redisplayed. This feature is useful within stored queries, where you can start a program or command without leaving SQL.

SAVING DATA TO TRS-XENIX FILES

There are two ways to send the output of a SQL query to a TRS-XENIX file. The first is from within SQL, using the keyword, **into**. The query:

```
select * from emp into employees/
```

puts the results of the query (in this case, the contents of the employee record type) into the file called "employees" in the current directory. This file contains the heading normally put at the beginning of each page of output. Note that the into clause must occur after the entire query block -- it is always the last clause. A longer query would appear as:

```
select ....
from ....
where ....
group by ....
having ....
```

```
order by ....
into filename/
```

The second way to save data to a TRS-XENIX file is to use a stored query and redirect the standard output to a file from the TRS-XENIX shell. For example, to send the results of the previous query into a file called "employees" without using the into keyword, put the query:

```
select * from emp/
```

into a file called "emps." Then, use the following TRS-XENIX command to save the results in the "employees" file:

SQL emps > employees <ENTER>

To use TRS-XENIX text processing filters, you probably do not want the descriptive heading. This can be eliminated with the keyword **lines**, which is used to set the number of lines per page. The default setting is 24 lines per page, the size of a standard screen. The setting controls the number of lines printed before a new heading is printed. Setting it to Ø suppresses the heading entirely. The following command at the sql> prompt suppresses the heading:

lines Ø <ENTER>

The field separator character can also be set as:

separator ':' <ENTER>

The default is the pipe symbol (|). Any single character, including tab and nonprinting characters, may be used.

LOADING DATA FROM TRS-XENIX FILES

Unify SQL lets you use the Database Load program to insert new records and modify existing ones in a database. The syntax uses an extension to the **insert** clause specifying the record type to load, the sequence of fields, and the name of the

TRS-XENIX file to use as the input file. This lets you load data from another database or applications system as well as shift data around within your Unify application.

Below is a sample file, named employees, containing employee records in the database load format. Note that this format requires separators between each field value.

57ØØ	Moehr	3Ø	clerk	64ØØ	95Ø.ØØ	Ø.ØØ
67ØØ	Colucci	4Ø	salesman	22ØØ	25ØØ.ØØ	3ØØØ.ØØ
58ØØ	Amato	4Ø	salesman	62ØØ	2ØØØ.ØØ	75Ø.ØØ
68ØØ	Fiorella	3Ø	clerk	57ØØ	8ØØ.ØØ	Ø.ØØ
59ØØ	Brown	6Ø	engineer	13ØØ	6ØØØ.ØØ	Ø.ØØ

You could load them into your database using the following example update statement. SQL automatically generates the specification file in this case, assuming that your input file contains all the fields in the record type in the default field order. (The default field order is the one returned by the query, select * from record.)

Note: A colon at the end of the **insert** clause is required, as shown in the example below.

Example: Load the records in the ASCII employees file into the database.

```
sql> insert into emp:
sql> from employees/
recognized update!
Loading record type: emp
Load complete  5 records inserted  Ø records updated
```

You can alo explicitly specify the order and fields in the input file. The following update illustrates the method. It assumes that a new record type, candidates, has been added to the schema.

Example: Save a subset of the emp record type to a TRS-XENIX file, and then load it back into the database. To get the save file in the load format, you must eliminate the header normally put at the beginning of the output. Use the **lines** keyword to do this. Once the data has been saved, load it back into a new record type, named candidates, that has the fields number, name, dept_no, and salary.

```
sql> lines ø
sql> select number, name, dept_no, salary
sql> from emp
sql> where dept_no = 7ø
sql> into candidates/
recognized query!
sql> insert into candidates (number, name, dept_no, salary):
sql> from candidates/
recognized query!
Loading record type: candidates
Load complete  2 records inserted  ø records updated
```

Reference

The previous two sections have presented examples of using Unify SQL. This section presents reference information for those who know how to use SQL but need quick information. The reserved keywords used by SQL are listed in "Keywords." "Error Messages" gives a summary of the error messages and possible corrective actions. It also presents diagrams of the formal language syntax of the complete Unify SQL language. Reference to these diagrams shows you how to construct any kind of SQL statement.

KEYWORDS

The following list of words have special meaning to SQL and cannot be used for record or field names. Queries using these words as record or field names result in syntax errors.

and
asc
avg
between
by
count
desc
edit
end
fields
from
group
having
help
in
insert
into
is
lines
max
min
not
or
order
records
restart
select
separator
start
sum
unique
unlock
where

ERROR MESSAGES

This section contains a summary of the error messages you might see while using SQL. They are listed alphabetically, with a brief explanation of the error's cause and an example (if possible) of the kind of statement that might produce the message.

There are two classes of errors -- errors in the pattern or structure of the keyword order of the statement (syntax errors) and errors in the meaning of the statement (semantic errors). Syntax errors prevent SQL from understanding what you are trying to say. Semantic errors result from a grammatically valid statement that SQL cannot process in a meaningful way. Either kind of error prevents you from getting a meaningful result.

Keep in mind that SQL cannot detect all possible semantic errors. Thus, it is possible to enter a query whose result does not mean what you might think. This is most likely to happen when using aggregate functions, when you try to list a field that is not a property of the group used to compute the aggregate. In these cases, SQL does not report an error, but the value(s) listed for the indicated field(s) does not coincide with the aggregate function value returned.

For example, suppose that you wanted to know the employees who make a salary equal to the average salary for all employees. You might try to use the following query to answer this question:

```
select name, avg(salary) from emp/
```

This does not result in an error, and in fact produces a name and the average salary for all employees. However, the name is **not** that of a person who makes that salary. In fact, there could be many employees who make the average salary, or none. To answer this question, you should use a nested query, such as:

```
select name, salary from emp
where salary =
select avg(salary)
from emp/
```

This is the same as asking the two questions -- "What is the average salary for all employees?" (the inner query) and "Which employees make that salary?" (the outer query).

***A 'having' clause cannot contain nested aggregate functions.

Explanation:
The having clause can only use simple aggregate functions. Nested aggregates do not make sense in this context.
Invalid Query:

```
select dept_no
from emp
group by dept_no
having avg(sum(salary)) > 5ØØØØ/
```

Correction:

```
select dept_no
from emp
group by dept_no
having sum(salary) > 5ØØØØ/
```

*** A 'having' clause must be preceded by a 'group by' clause.

Explanation:
The having clause calculates the values of aggregate functions based upon the groups of records specified by the group by clause, so a group by is required.
Invalid Query:

```
select job
from emp
having avg(salary) > 2ØØØ/
```

Correction:

```
select job
from emp
group by job
having avg(salary) > 2ØØØ/
```

*** a 'having' clause requires an aggregate function.

Explanation:
The having clause selects based upon an aggregate property

```
    of a group, so you must use an aggregate function.
Invalid Query:
    select job, salary
    from emp
    group by job
    having salary > 15ØØ/
Correction:
    select job, avg(salary)
    from emp
    group by job
    having avg(salary) > 15ØØ/
```

***A literal tuple contains the wrong number of items.

```
Explanation:
    The field specification list and every literal tuple must
    contain the same number of items.  Otherwise, SQL does not
    know what to do with the "extra" value or "missing" field.
Invalid Query:
    select name
    from emp
    where <job, salary> = <'clerk*'>/
Correction:
    select name
    from emp
    where <job, salary> = <'clerk*', 95Ø>/
```

*** Aggregate functions are not allowed in the 'where' clause.

```
Explanation:
    Aggregate functions can only be used in having and select
    clauses.  Rewrite the query using a having clause if you
    want to select based on an aggregate function.
Invalid Query:
    select job
    from emp
    where avg(salary) > 1ØØØ/
Correction:
    select job
    from emp
```

```
    group by job
    having avg(salary) > 1ØØØ/
```

*** Aggregate function imbalance in the 'select' clause.

Explanation:
    All the aggregate functions within a single query must be nested to the same level. If you want to see both answers, perform two queries.

Invalid Query:

```
    select avg(salary), sum(avg(salary))
    from emp/
```

Correction:

```
    select avg(salary)      or      select sum(avg(salary))
    from emp/                       from emp
                                    group by dept_no/
```

*** Aggregate functions may not be nested without a 'group by' clause.

Explanation:
    Since an aggregate function calculates a single value for a group of values, it does not make sense to apply an aggregate to a single value. This happens when you try to nest aggregate functions without using a group by clause.

Invalid Query:

```
    select sum(avg(salary))
    from emp/
```

Correction:

```
    select sum(avg(salary))
    from emp
    group by job/
```

*** Aggregate functions nested too deeply in the 'select' clause.

Explanation:
    Aggregate functions may only be nested to two levels.

Invalid Query:

```
    select max(sum(avg(salary)))
```

```
    from emp/
Correction:
    select max(sum(salary))
    from emp
    group by dept_no/
```

*** An aggregate function is required when using a 'group by' clause.

```
Explanation:
    A group by clause groups records only for evaluation of
    aggregate functions, so it does not make sense to group
    records without calculating an aggregate function.
Invalid Query:
    select job, salary
    from emp
    group by job/
Correction:
    select job, min(salary)
    from emp
    group by job/
```

*** An expression cannot be compared with a literal tuple list.

```
Explanation:
    The result of an expression is a single value, while a
    literal tuple contains several values.  SQL does not know
    what to do with the "extra" values in the comparison.
Invalid Query:
    select name
    from emp
    where salary+commission is in
    (<25ØØ, 'salesman*'>,
    <8ØØ, 'clerk*'>,
    <11ØØ, 'programmer*'>)/
Correction:
    select name
    from emp
    where salary+commission is in <25ØØ, 8ØØ, 11ØØ>/
```

*** Can't create 'xxxx'.

Explanation:
SQL cannot create the file name "xxxx." Check the read and write permissions of the current directory to make sure that the current TRS-XENIX user can write in the directory.

*** Fatal error

Explanation:
SQL has discovered an uncorrectable internal error. The error message preceding this one indicates the nature of the error. Try rewriting the query using a different method.

*** Field: ffff requires a password, use 'unlock.'
*** Field: xxxx.ffff requires a password, use 'unlock.'

Explanation:
The indicated field in either the select, where, or having clause has been protected with Field Level Security. You must unlock the field before you can list it.

*** Insufficient memory for sort line.

Explanation:
There are too many elements in the group by or order by clause. Use fewer sort elements.

*** Invalid 'group by' clause.

Explanation:
The syntax of the group by clause is incorrect. This could be caused by many different errors. In the following query, the problem is that you cannot group by computed fields.
Invalid Query:

```
select avg(salary)
from emp
group by (salary+commission)/
```

```
Correction:
    select avg(salary)
    from emp
    group by salary, commission/


*** Invalid 'having' clause.

Explanation:
    The syntax of the having clause is incorrect.  This could
    be caused by many different errors.  In the following query,
    the problem is that you must use a boolean expression (an
    expression using one of the comparison operators) in a
    having clause.
Invalid Query:
    select dept_no
    from emp
    group by dept_no
    having max(avg(salary))/

Correction:
    select dept_no
    from emp
    group by dept_no
    having avg(salary) = select max(avg(salary))
    from emp
    group by dept_no/


*** Invalid 'select' list syntax.

Explanation:
    The syntax of the select clause is incorrect.  This could
    be caused by many different kinds of errors.  In the
    following query, the fields to be selected are enclosed in
    broken braces, which is not required.
Invalid Query:
    select <name, job>
    from emp/
Correction:
    select name, job
    from emp/
```

*** Invalid 'where'clause.

Explanation:
    The syntax of the where clause is incorrect. This could be caused by many different kinds of errors. In the following query, the same problem occurs as in the previous invalid having clause -- the expressions lacks a comparison operator with which to perform a comparison.
Invalid Query:
```
    select name
    from emp
    where salary/
```
Correction:
```
    select name
    from emp
    where salary > commission/
```

*** Invalid date.

Explanation:
    The format of a DATE constant is incorrect. DATE constants are in the form MM/DD/YY. MM is the month (1-12), DD is the day (1-31), and YY is the year.

*** Invalid field specification: xxxx.ffff

Explanation:
    The indicated field specification does not make sense -- "xxxx" is the record, and "ffff" is the field. In the following query, the record type is misspelled.
Invalid Query:
```
    select epm.name
    from emp/
```
Correction:
```
    select emp.name
    from emp/
```

*** Invalid field: ffff

Explanation:
The field name "ffff" is unrecognized. Check to make sure you are using the long field name and that you have not misspelled it.
Invalid Query:
```
select ename
from emp/
```
Correction:
```
select name
from emp
```

*** Invalid format for field "ffff" record "xxxx".

Explanation:
A modification has been made to the key field without rebuilding the hash table.
Correction:
Reconfigure the database and rebuild the hash table.

*** Invalid query block.

Explanation:
The query expression is unrecognizable. This could be caused by many different kinds of errors. In the following query, the keyword, select, is misspelled.
Invalid Query:
```
sleect name
from emp/
```
Correction:
```
select name
from emp/
```

*** Invalid record type specification in the 'select' clause: xxxx

Explanation:
The record type "xxxx" used before ".*" is unknown. The

problem is probably a misspelling.

```
Invalid Query:
    select epm.*
    from emp
Correction:
    select emp.*
    from emp/
```

*** Invalid record type: xxxx

Explanation:
The record type "xxxx" used in the query is unrecognized. The problem is probably a misspelling.

```
Invalid Query:
    select name
    from epm/
Correction:
    select name
    from emp/
```

*** Invalid time.

Explanation:
The format of a TIME constant is incorrect. TIME constants are in the form HH:MM. HH is the hour (Ø-32), and MM is the minute (Ø-59).

*** ffff is an invalid field name for record type xxxx.

Explanation:
The field specified in the unlock command is not contained in the record type you specified. Check your spelling and the list of valid field names for the particular record type.

*** ffff is an invalid field name.

Explanation:

The indicated field in the field specification list for the insert statement is incorrect or has been misspelled.

*** xxxx is an invalid file name.

Explanation:
The TRS-XENIX file named "xxxx" is either nonexistent or unreadable. You may have misspelled the name, or the file has a mode that does not permit you to read it. Either can be checked with the TRS-XENIX command ls(1).

*** pppp is an invalid password for xxxx.ffff

Explanation:
The password you specified to unlock the field is incorrect.

*** xxxx in an invalid record type.

Explanation:
The record type indicated does not exist in the database. You probably misspelled the name.

*** No help is available on this subject.

Explanation:
Either you misspelled the keyword for which you wanted help, or the help file for the keyword is missing or unreadable. The Unify environment variable may be set incorrectly, and so the help dictionary cannot be found.

*** Not enough memory

Explanation:
The query is too long or is nested too deeply for the hardware on which you are running. Simplify the query, or break it up into smaller queries.

*** SQL: Can't open xxxx

Explanation:
SQL was started from the shell command line to execute the script in the file named "xxxx." The current TRS-XENIX user cannot open this file. Change the file permission mode to permit the user to read the file.

*** Syntax error...

Explanation:
The current query contains a syntax (grammatical) error. Another error message should appear immediately following this one, explaining the error in more detail.

*** The 'from' clause is missing or invalid.

Explanation:
Every query requires a from clause. Either the from clause is missing or its syntax is incorrect. This could be caused by many different kinds of errors. In the following query, the keyword, "from," is misspelled.

Invalid Query:
```
select name
form emp/
```
Correction:
```
select name
from emp/
```

*** The correct usage is: SQL [filename]

Explanation:
SQL was started from the shell command line with an incorrect number of parameters.

*** The field 'ffff' is not in record type 'xxxx'.

```
Explanation:
    The field 'ffff,' while a valid database field, is not in the
    record type 'xxxx.'
Invalid Query:
    select emp.location
    from emp/
Correction:
    select dept.location
    from dept/


*** The file 'xxxx' already exists.

Explanation:
    The query directed its output to a file that already exists,
    using the into clause.  SQL does not overwrite an existing
    file.  Either remove the file, or direct the output to a file
    with a different name.


*** Type conversion error.
    Left type is XXXX, right type is XXXX, operator is XXXX.

Explanation:
    The types of two operands do not agree, and one type cannot
    be converted into the other type. You probably used the wrong
    field name.
Invalid Query:
    select name
    from emp
    where name = 2ØØØ/
Correction:
    select name
    from emp
    where salary = 2ØØØ/


*** Unable to open xxxx

Explanation:
    SQL was started from the command line, taking its input from
```

a script named "xxxx." This file is inaccessible. Check that the current TRS-XENIX user has permission to read the file.

*** Unrecognized sql command

Explanation:
The first keyword on the SQL command was not recognized. The problem is probably a misspelling.

*** Warning: 'group by' and 'order by' in the same query.
The 'order by' will be ignored.

Explanation:
The group by and order by clauses cannot be used in the same query. This message is a warning that the order by will not be performed. The query will still be executed.

*** Warning: 'unique' may not be used with aggregate functions.
The 'unique' specification will be ignored.

Explanation:
The unique specification may not be used in conjunction with any of the aggregate functions min, max, sum, avg, or count. The query is still performed, but duplicate rows are not eliminated.

*** on or about line nn

Explanation:
This message is sent to the standard error output when you run a query from a script and a syntax error is detected. It indicates the approximate location of the error.

FORMAL GRAMMAR SYNTAX

The Unify SQL grammar is composed of three basic parts, each with its own structure and function. This section explains the function of each part of the grammar, and the following subsections detail the structure of each part.

The first and largest part of the grammar is the query language. The query language lets you ask questions about the information in your database by asking "what" and letting SQL find "how" to get the information.

The next part of the grammar is the command language. The command language has many capabilities that make the interactive development and execution of query statements easier. Some of the capabilities include editing the current SQL statement, starting and restarting queries from script files, unlocking fields protected by field level security, and executing TRS-XENIX shell commands. The last part of the grammar is SQL help. SQL help lets you quickly review how a particular command is used and list the valid record and field names in the current database.

The following notation is used throughout this section to describe the structure of the grammar:

lowercase words	Place holders for words, numbers, or expressions that you define. Examples are record and field names or constants.
[ ]	Indicates that the item enclosed in square brackets is optional and may be omitted.
| |	Vertical bars enclosing a stacked list of options means that only one of the options can be chosen.
...	Indicates that the immediately preceding item may be repeated any number of times.

Query Statements

A SQL query statement consists of one or more "query blocks," terminated with a slash (/). A query block always starts with a select clause that indicates which fields or expressions are to be retrieved. It continues with a from clause that indicates the record types from which the field are to be retrieved, and then includes none, one, or more optional clauses. The clauses must occur in the order below, even though some of them may be omitted in a specific query. The dots in the illustration stand for the continuation of the clause.

```
select ....
from ....
[ where .... ]
[ group by .... ]
[ having .... ]
[ order by .... ]
[ into .... ]
```

Select Clause

```
select [ unique ] | *           |, | record.*      |, ...
                  | record.*    |  | record.field  |
                  | record.field|  | field         |
                  | field       |  | constant      |
                  | constant    |  | count (*)     |
                  | count (*)   |  | min (expr)    |
                  | min (expr)  |  | max (expr)    |
                  | max (expr)  |  | sum (expr)    |
                  | sum (expr)  |  | avg (expr)    |
                  | avg (expr)  |  | expr          |
                  | expr        |
```

The select clause lets you specify the fields or expressions to retrieve from your query. Every query must start with a select clause.

unique	The unique keyword eliminates duplicate rows from the query result. If every column in two or more rows matches, only one row is displayed in the result. Use of this keyword sorts the query output.
*	The asterisk selects all fields of all record types named in the from clause. If you use an asterisk, no other fields or expressions are allowed in the select clause.
record.*	This is the name of a record type in the database, followed by a period and an asterisk. The result selects all the fields from the indicated record type.
record.field	This is the name of a field, preceded by the name of the record type in which it occurs. If the record has been assigned a label in the from clause, the label can be used in place of the record name. You must qualify a field in this way when you are using a field name that occurs in two or more different record types appearing in the from clause. Without the qualification, it is not clear which record type should be used to get the field.
field	The name of a field that occurs in one of the records in the from clause.
constant	A constant value of NUMERIC, STRING, DATE, TIME, AMOUNT, or FLOAT type. The correct type of constant to use in any given situation is determined by the type of the field or expression with which you are comparing the constant. STRING constants are enclosed in single quotation marks (') to distinguish them from field names.
count(*) min (expr) max (expr)	These are the built-in SQL aggregate functions. Using an aggregate function implies that a group by is to be performed. If you did not specify

avg (expr)
sum (expr)

the group explicitly, the entire query result is treated as a single group. It also implies that you only selected fields or expressions whose values are common to all records in the group. The count(*) function counts the number of records in each group that matches the query selection criteria. The min, max, and avg functions calculate the minimum, maximum, and average values of the indicated expression for the records in each group that match the query selection criteria. The sum function adds up the indicated expression for the records in each group that match the query selection criteria. These functions may be applied to NUMERIC, AMOUNT, FLOAT, DATE, and TIME fields, although the results on DATE and TIME fields are probably not meaningful. Aggregate functions can also be nested to achieve expressions such as "avg(count(*))" and "min(avg(salary))."

expr — An arithmetic expression, which may be composed of fields, constants, and aggregate functions, connected with the operators +, -, *, and /. Parentheses indicate the order in which the operations are to be performed.

From Clause

```
from | record [label] |, ...
```

The from clause lets you specify the record types to be used in the query. You can list any number of records, in any order, and SQL figures a way to get the data.

record — The name of a record type in the current database.

label — A label used to identify the record type later on in the query. Labels are used in self-join and variable queries.

Where Clause

```
where [ not ]

| |field         | | =     | |field                          |        | |and|
| |record.field  | | ^=    | |record.field                    |        | |or | ...
| |constant      | | >     | |constant                        |        |
| |expr          | | >=    | |expr                            |        |
                   | <     | |select .... [;]                |        |
                   | <=    |  <constant, ...>               |        |
                   | in    | |(<constant, ...>, ...)        |        |
                   |is in  |                                |        |

| |field         | between |field         | and |field         |        |
| |record.field  |         |record.field  |     |record.field  |        |
| |constant      |         |constant      |     |constant      |        |
| |expr          |         |expr          |     |expr          |        |

| <|field       |, ...> | in    | |select .... [;]       |        |
|  |record.field |       |is in  | |<constant, ...>       |        |
                         | =     | |(<constant, ...>, ...)|        |
```

The where clause lets you specify selection criteria to select or reject individual records in the query result. The notation above means that a where clause is composed of one or more boolean expressions connected with "and" and "or." Each boolean expression can take one of the three forms indicated, either using the comparison operators, the between-and operators, or the set equality operators. If a where clause contains multiple boolean expressions connected with either "and" or "or," square brackets can be used to indicate the expression to be evaluated first.

field — The name of a field that occurs in one of the records in the from clause.

record.field — The name of a field, preceded by the name of the record type in which it occurs. If the record has been assigned a label in the from clause,

the label can be used in place of the record name. You must qualify a field in this way when you use two or more different record types (each containing fields with the same name) in the from clause. Without the qualification, SQL does not know from which record type to get the field.

constant — A constant value of NUMERIC, STRING, DATE, TIME, AMOUNT, or FLOAT type. The correct type of constant to use in any given situation is determined by the type of the field or expression with which you are comparing the constant. STRING constants are enclosed in single quotation marks (') to distinguish them from field names.

expr — An arithmetic expression that may be composed of fields and constants, connected with the operators +, -, *, and /. Parentheses indicate the order in which the operations are performed.

select .... — Indicates that a query block may be used in this part of the statement. Refer to the beginning of this section for more information.

<constant, ...> — Indicates that a list of constants (a "literal tuple") may be used instead of a single value when comparing for equality with the left side of a boolean expression. A literal tuple is a list of constants, separated by commas and enclosed in angled brackets (< >). A common usage of this kind of expression is:

```
field is in <c1, c2, c3>
```

This is the same as the more cumbersome expression:

```
field = c1 or field = c2 or field = c3
```

(<constant, ...>, ...) Indicates that a list of literal tuples may be used when comparing for equality with the left side of a boolean expression. Each tuple is constructed as above, with commas separating the tuples and the entire list enclosed in parentheses. A common usage of this kind of expression is:

```
<f1, f2> is in (<c1, c2>, <c3, c4>)
```

This is the same as:

```
[f1 = c1 and f2 = c2] or
[f1 = c3 and f2 = c4]
```

See "Set Inclusion" in the first half of this chapter for examples of how literal tuples may be utilized.

`=`
`^=`
`>`
`>=`
`<`
`<=`

The standard comparison operators -- equal, not equal, greater than, greater than or equal to, less than, and less than or equal to.

`in`
`is in`

These two operators are the same as the standard equal sign. They are provided so that queries involving literal tuples read more naturally. The regular equal sign has the same effect.

`between -- and`

A single operator provided to make ranges easier to specify. The normal way of specifying a range in a query language is as follows:

```
field <= c1 and field >= c2
```

Using this operator, the expression reads:

```
field between c1 and c2
```

This is a more natual way of stating range selections.

and — The standard conjunction operator. The value of two simple boolean expressions connected with "and" is true if both simple expressions are true and false if either simple expression is false.

or — The standard disjunction operator. The value of two simple boolean expressions connected with "or" is true if either simple expression is true and false if both simple expressions are false.

Group By Clause

```
group by | field        |, ...
         | record.field |
```

The group by clause lets you sort records into groups so that you can apply an aggregate function to each group. A group by clause used alone has no meaning without an aggregate function. If you don't specify a group with this clause and use an aggregate function in the select clause, the entire query result is treated as a single group. The use of this clause also implies that each field or expression in the select clause has the same value for each record in the same group. Although you do not receive a syntax error if you list fields or expressions whose values vary for each record in the group, neither do you receive a meaningful query result.

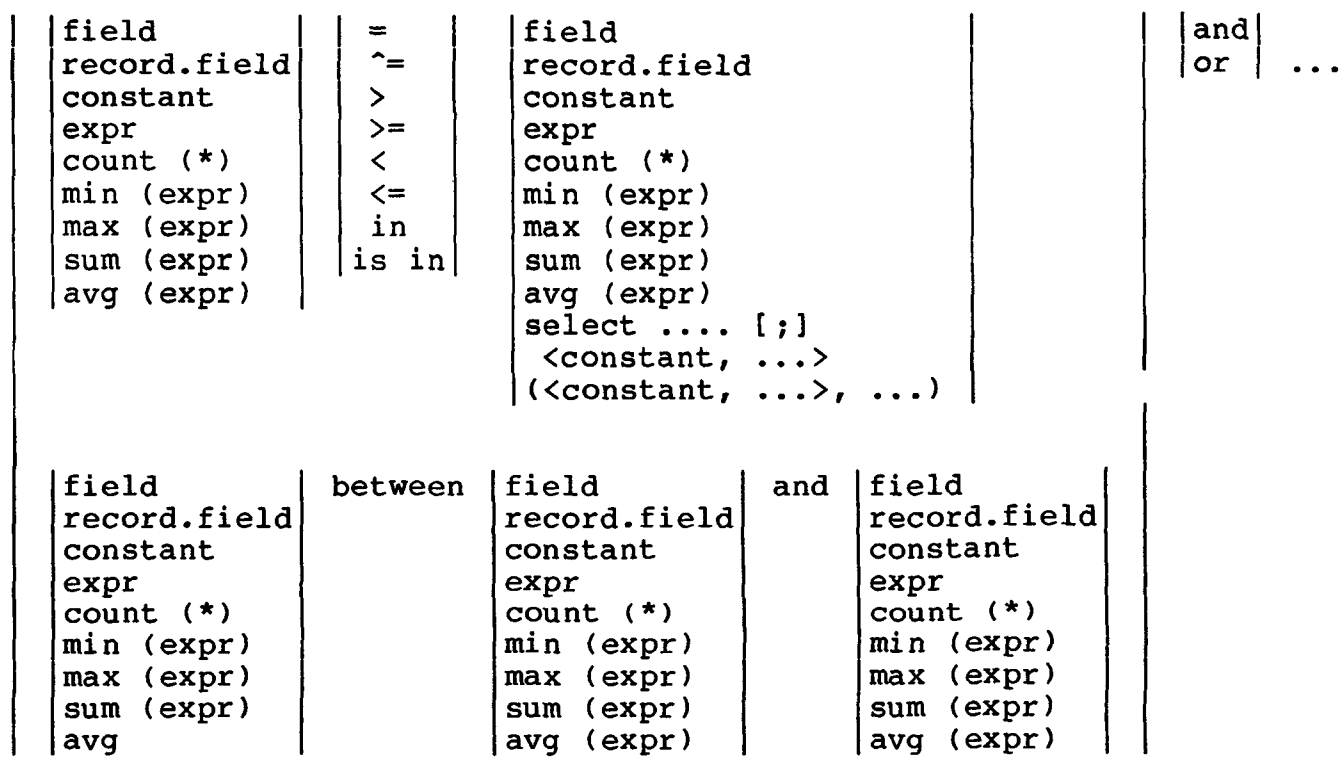
field — The name of a field that occurs in one of the records in the from clause.

record.field — The name of a field, preceded by the name of the record type in which it occurs. If the record has been assigned a label in the from clause, the label can be used in place of the record name. You must qualify a field in this way when you are using a field name occurring in two or more

different record types in the from clause. Without the qualification, SQL does not know the record type from which to get the field.

Having Clause

having



A having clause consists of one or more boolean expressions connected with "and" and "or". Each boolean expression can take one of the two forms indicated, either using the comparison operators or the "between-and" operator. If a having clause contains multiple boolean expressions connected with "and" or "or," use square brackets to indicate the expression to be evaluated first.

This clause lets you select some of the groups formed by the group by clause and reject others, based upon the value of an aggregate function. The having clause must always contain at least one aggregate function and be preceded by a group by clause.

field	The name of a field that occurs in one of the records in the from clause.
record.field	The name of a field, preceded by the name of the record type in which it occurs. If the record has been assigned a label in the from clause, the label can be used in place of the record name. You must qualify a field in this way when you are using a field name occurring in two or more different record types in the from clause. Without the qualification, SQL does not know the record type from which to get the field.
constant	A constant value of NUMERIC, STRING, DATE, TIME, AMOUNT, or FLOAT type. The correct type of constant to use is determined by the type of the field or expression with which you are comparing the constant. STRING constants are enclosed in single quotation marks (') to distinguish them from field names.
expr	An arithmetic expression, which may be composed of fields, constants, and aggregate functions, connected with the operators +, -, *, and /. Parentheses indicate the order in which the operations are performed.
count(*) min (expr) max (expr) avg (expr) sum (expr)	The built-in SQL aggregate functions. When you use an aggregate function, it implies that a group by will be performed. With a having clause, you must specify the group explicitly. It also implies that you are only selecting fields or expressions whose values are common to all records in the group. The count (*) function counts the number of records in each group that matches the query selection criteria.

The min, max, and avg functions calculate the minimum, maximum, and average values of the indicated expression for the records in each group that match the query selection criteria. The sum function adds up the indicated expression for the records in each group that match the query selection criteria. These functions may be applied to NUMERIC, AMOUNT, FLOAT, DATE, and TIME fields, although the results on DATE and TIME fields are probably not meaningful. Aggregate functions can also be nested to achieve expressions such as "avg (count(*))" and "min(avg(salary))."

select .... Indicates that a query block may be used in this part of the statement. Refer to the beginning of this section for more information.

<constant, ...> Indicates that a list of constants (a literal tuple) may be used instead of a single value when comparing for equality with the left side of a boolean expression. A literal tuple is a list of constants, separated by commas and enclosed in angled brackets (< >). A common usage of this kind of expression is:

```
field is in <c1, c2, c3>
```

This is the same as:

```
field = c1 or field = c2 or field = c3
```

(<constant, ...>, ...) Indicates that a list of literal tuples may be used when comparing for equality with the left side of a boolean expression. Each tuple is constructed as above, with commas separating the tuples and the entire list enclosed in parentheses. A common usage of this kind of expression is:

```
<f1, f2> is in <c1, c2>, <c3, c4>)
```

This is the same as:

```
[f1 = c1 and f2 = c2] or
[f1 = c3 and f2 = c4]
```

See "Set Inclusion" in the first half of this chapter for examples of how literal tuples may be utilized.

`=`
`^=`
`>`
`>=`
`<`
`<=`

These are the standard comparison operators -- equal, not equal, greater than, greater than or equal to, less than, and less than or equal to.

`in`
`is in`

These two operators are the same as the standard equal sign. They are provided so that queries involving literal tuples read more naturally. The regular equal sign has exactly the same effect.

`between -- and`

A single operator provided to make ranges easier to specify. The normal way of specifying a range in a query language is:

```
field <= c1 and field >= c2
```

Using this operator, the expression reads:

```
field between c1 and c2
```

This is a more natural way of expressing a range selection.

`and`

The standard conjunction operator. The value of a compound boolean expression connected with "and" is true if both simple expressions are true and false if either simple expression is false.

`or`

The standard disjunction operator. The value of a compound boolean expression connected with "or"

is true if either simple expression is true and false if both simple expressions are false.

Order By Clause

```
order by | field        [ |asc | ] |, ...
         | record.field   |desc|   |
```

This clause lets you sort the output of a query. You can specify ascending or descending sorts on each field or omit the specification entirely. The default sort order is ascending.

field	The name of a field that occurs in one of the records in the from clause.
record.field	The name of a field, preceded by the name of the record type in which it occurs. If the record has been assigned a label in the from clause, the label can be used in place of the field name. You must qualify a field in this way when you are using a field name that occurs in two or more different record types in the from clause. Without the specification, SQL does not know from which record type to get the field.
asc desc	Indicate the order in which the preceding field is to be sorted. If you want descending order, use desc. If you want ascending order, you may specify asc for clarification or leave this specification off since the default sort order is ascending.

Into Clause

into filename

This clause sends the query results to a TRS-XENIX file instead of the screen. This is useful if you want to use the data in another program or load it into another database.

filename	The name of a TRS-XENIX file in which to create and store the results of the current query. An error occurs if the file already exists.

Commands

Unify SQL includes a number of commands that are not strictly part of the language. These commands let you edit the current SQL statement, restart the current statement, start SQL statements stored in text files, unlock fields protected by passwords, set the number of lines per page for the column headings, change the standard field separator, execute TRS-XENIX commands, and exit SQL. The following sections explain each of these commands in more detail.

Edit Command

edit

The current SQL statement is stored in a temporary file in the directory, /tmp. This command lets you edit it to correct mistakes in syntax or modify it to produce a different result.

Restart Command

restart

This command lets you execute the current SQL statement stored in the temporary file. When you type in a new statement,

it replaces the old one, becoming the new current statement. This command is most often used after editing the current statement.

Start Command

start filename

This command executes a SQL statement stored in the TRS-XENIX filename file. Executing a statement in this way does not cause it to become the current statement. The current statement is the last one you entered.

filename	The name of a TRS-XENIX file that contains a SQL statement. The statement is executed, and the results are displayed on the screen.

Unlock Command

unlock record.field, password, .../

This command unlocks database fields that have been protected by Field Level Security. This command must be terminated by a slash (/).

record.field	The name of the database field you want to unlock. It must be preceded by the name of the record of which it is a part.
password	The password that was entered in Field Level Security to lock this field.

Lines Command

lines number

This command changes the count at which a new heading is printed. The default is set to 24 lines, the size of most screens. Every 24 lines, a new header is printed.

number — A number between Ø and 32767 that gives the line count at which to print a new heading. If the number is Ø, no heading is printed. Otherwise, a heading is printed at the start and after every "number" lines of output.

Separator Command

separator 'character'

This command lets you change the field separator used by SQL to indicate where one field ends and the next starts. The default is the vertical bar (|).

character — A single character that is the new field separator. This can be any character you want, including nonprinting control characters. If you want to load an ASCII file using the insert statement, the ASCII file must use the same field separator throughout, and SQL's field separator must match the one in the file you want to load.

TRS-XENIX Commands

! TRS-XENIX command

Any command preceded by an exclamation point (!) is passed on to the TRS-XENIX shell for execution. When the command is finished, SQL regains control.

End Command

end

To exit SQL and return to either the menu handler or the TRS-XENIX shell (depending upon how you started), type **end** <ENTER> at the sql> prompt.

Help Commands

A SQL help command consists of one or two keywords, the second indicating the subject about which you would like to see help information. Help is intended to serve as a reminder about the syntax of any particular statement or command. Once you know how to use SQL, you can use help as a quick reference instead of referring to the manual. The following sections describe the ways in which you can use SQL help.

Help Command

help [keyword]

This statement displays help information about how to use SQL. If you type **help** <ENTER>, you see a general message about how to use the help features. If you type **help keyword** <ENTER>, you see help information about that specific keyword.

keyword — One of the keywords used by SQL. If there is help documentation about that keyword, it is displayed. A list of the keywords is in "Keywords."

Records Command

records

This command lists the names of all the record types you can use in constructing SQL statements. There are no options for this command.

Fields Command

fields record type

This command lists all the fields for the indicated record type, their types, and their lengths.

record type	The name of a record type in the current database. If you do not know any record types, use the records help command.

LST -- LISTING PROCESSOR

The Unify Listing Processor, LST, is a selection and formatting language that produces listings containing totals and subtotals that are sorted from a Unify database. LST provides a simple alternative to SQL, when the full query capabilities of this tool are not required but custom formatted reports are required. LST has two major components -- the selection processor, which lets you choose records from a file based on selection criteria, and the listing processor, which lets you sort, format, and total the selected records.

The selection processor is an easy to use relational query language that lets you make inquiries into a Unify database. You are able to combine multiple record types in a single query by using a powerful "natural join" capability. The syntax of the query is not dependent on the different access methods allowed in a Unify database. This means that queries can usually be left unchanged when either the physical or logical database description is modified.

The result of a selection processor query is a "selection file" that contains the addresses of all the records that satisfy the selection. Another name for a selection file is "subset." As with ENTER, the records chosen by the selection processor can be used by the listing processor, which allows for custom output. Alternatively, for maximum flexibility, a user program can also process the selection file.

The listing processor is a general purpose tool for producing custom formatted listings from selection files. It is easy to use, yet powerful enough to fulfill most listing needs. Although the listing processor facilitates the design of custom report formats, default values provided for almost every option simplify the report design process.

At the simplest level, you specify the fields to list, and the program assigns columns and headings and then formats the listing. At the most complex level, you can specify in which column and line to print the field or expression and what format to use. You can also define totals, subtotals, and sort order. In

addition, you are able to specify multi-line column and listing headings.

Selection files are produced in three different ways: using the selection processor, using the query by forms capability of ENTER, or using a program. The selection file produced by each of these three methods carries with it the record type selected. This record type is then used by the listing processor to produce the final output.

Selecting Records

Use commands, such as SELECT, CALL, COPY, and LIST, to select records based on specific criteria. These records and their addresses are then placed in a selection file that can be passed to the Listing Processor.

RUNNING THE SELECTION PROCESSOR

You can run the selection processor either from the menu handler, from the shell, or from a shell script. When run from the menu handler or an interactive shell, the selection processor issues the "*" prompt and continues to accept commands until the "end" command is entered. To execute a query in a non-interactive manner, use the following syntax:

LST {query-script}

When running the selection processor in this manner, it executes the commands in the query-script until end of file is reached.

SELECTION SYNTAX

The following symbols are used to describe the syntax of the selection processor:

{ } The braces indicate a value or variable to be filled in by the user. For example, {file name} means to enter the desired file name at that point. The other values used are: "expr" for an expression (refer to the next section, "Expressions"), "subset" for a named selection file created by the selection processor, "field" for a database field name, "password" for a password assigned to a field using field level security, and "string" for any alphanumeric string.

[ ] The brackets indicate that an entry (or entries) is optional.

.... The dots indicate that the previous syntax can be repeated any number of times.

| Vertical bars enclose options, and you must choose one of the options to create a valid query.

Expressions

The term "expr" is used by the selection processor to refer to an algebraic combination of fields, operators, and constants. The syntax referred to by "expr" appears below:

expr	+	expr
field	-	field
number	*	number
string	/	string
ref-field	=	substr-spec
	!=	ref-field
	<=	
	>=	
	<	
	>	
	substr	
	and	
	or	

You can surround any expression or sub-expression with parentheses to indicate precedence. Without parentheses, the operators are grouped in the following priority: highest [* /], medium [+ - substr], lowest [< > <= >= =]. Two elements of type "string" can be concatenated using the "+" operator. Other than that, only the relational operators and substr apply to these elements. "Substr" applies only to elements of type "string." The "substr-spec" is of the form "start-end", where "start" is the starting character position (indexed from 1) and "end" is the ending character position. For example, the first 1Ø characters of a string would be "1-1Ø."

Reference fields, "ref-field", have the following syntax:

field.field....

Reference fields specify fields in record types other than the record type being selected. The first field should be the key of the record type containing the second field. Each field is used to access the next field, using the field names defined as explicit references within Schema Maintenance, and so on through the final field, which is the value of the element.

SELECT SELECT

SYNTAX

```
select | {subset}    |  [into {string}] where {expr} end
       | {file name} |
```

USAGE

Use this command to select a portion of a file or current subset. The results of the selection are placed into a subset which can either be passed to the listing processor, a program, or used in a subsequent select.

If the name following "select" is not a previously selected subset, the selection is made from the entire set of records in the specified file. Otherwise, the selection is made from the specified subset. A record is selected when the result of the expression is non-zero for that record.

By default, the results are placed into a subset with the same name as the originating subset or file. If the "into" syntax is used, the output subset name is specified by the string.

REMOVE **REMOVE**

SYNTAX

remove {subset}

USAGE

Use this command to remove a previously selected subset. Since select only selects data from the database, if there is not a subset by the same name, it is a good idea to remove subsets after they have been used. You can, however, use **call** to change the subset name.

CALL **CALL**

SYNTAX

call {subset} {string}

USAGE

Use this command to change the name of an existing subset. The specified subset is renamed "string." The renamed subset retains all of the characteristics of the original one.

COPY COPY

SYNTAX

copy {subset} {string}

USAGE

Use this command to make a copy of an existing subset. It is useful for performing a reselect while maintaining the results of the first selection. The named subset is copied into a new subset with the name "string." The copy has all the characteristics of the original subset.

LIST **LIST**

SYNTAX

list {generic-subset-name}

USAGE

Use this command to list the existing subsets. The "generic-subset-name" may have the same characteristics of a TRS-XENIX file specification. Within the generic-subset-name, "*" matches any number of characters, "?" matches any single character, and "[..]" matches any single character in the specified set. For each subset which matches the specification, the name, the Unify record type, and the number of records are listed.

UNLOCK UNLOCK

SYNTAX

unlock {field} {password}

USAGE

Use this command to allow a secure field to be read by the selection processor. (See "Field Level Security" in Chapter 3 of this manual.) You can specify either the read or write password for the file. Any attempt to access a field, which is locked before you enter this command, results in an error message.

REPORT REPORT

SYNTAX

report {subset}

USAGE

This command invokes the listing processor. The listing processor is a general tool for creating simple listings from the selected records. The subset name is the selection file on which the listing processor is to operate.

Listing Records

Once you have selected a set of records, using either the selection processor, ENTER, or a program, you can use the listing processor to produce a formatted report. The listing processor is easy to use, yet powerful enough to fulfill many reporting needs.

In its simplest form, the listing processor lets you specify the fields to list, and LST assigns the columns, headings, and formats. At the most complex level, you can specify in which column and line each field or expression is to be printed and what format is to be used. Totals, subtotals, and sort order can also be defined. In addition, you can specify multi-line column and report headings.

RUNNING THE LISTING PROCESSOR

There are several ways to run the listing processor. The simplest way is to use the "report" command in the selection processor. The listing processor issues the "-" prompt and accepts commands interactively. A final "end" command terminates the program. The second way to use the listing processor is as a "canned" report registered with an ENTER screen. To do this, use a text editor to place listing commands in a file, then register the name of the text file with the ENTER screen using ENTER Screen Registration (chapter 5 in this manual). The third alternative is to run the listing processor directly from the shell or shell script. The syntax for this method is as follows:

```
LST -r {selection-file} {report-script}
```

-r indicates that the second argument passed to this function is a report script name.

When invoked by using the above syntax, the listing processor continues to execute the commands in the report-script file until end of file is reached.

LISTING PROCESSOR SYNTAX

The following symbols are used to describe the syntax of the listing processor:

{ } The braces indicate a value or variable to be filled in by the user. For example, {file name} means to enter the desired file name at that point. The other values used are: "expr" for an expression, "field" for a database field name, "format" for a format specification for numeric fields, "name" for a column heading to be associated with a field or expression, "password" for a password assigned to a field using field level security, and "string" for any string of characters.

[ ] The brackets indicate that an entry (or entries) is optional.

.... The dots indicate that the previous syntax can be repeated any number of times.

| Vertical bars enclose options, and you must choose one of the options to create a valid query.

Expressions

The listing processor uses the term "expr" to refer to an algebraic combination of fields, operators, and constants. The syntax referred to by "expr" appears below:

expr	+	expr
field	-	field
number	*	number
string	/	string
ref-field	=	substr-spec
	!=	ref-field
	<=	
	>=	
	<	
	>	
	substr	
	and	
	or	

You can surround any expression or sub-expression with parentheses to indicate precedence. Without parentheses, the operators are grouped as follows: highest [* /], medium [+ - substr], and lowest [< > <= >= =]. Two elements of type "string" can be concatenated using the "+" operator. Other than that, only the relational operators and substr apply to strings. The "substr-spec" is of the form "start-end", where start is the starting character position (indexed from 1) and end is the ending character position. For example, the first 1Ø characters of a string would be "1-1Ø."

Reference fields, "ref-field", have the following syntax:

field.field....

Reference fields specify fields in record types other than the record type being listed. The first field should be the key of the record type containing the second field. Each field is used to access the next field, using the field names defined as explicit references within Schema Maintenance, until reaching the final field, which is the value of the element.

LIST LIST

SYNTAX

```
list [column {number} [line {number}]]
     [{name}] {expr} [using {format}]
     [,[{name}] {expr} [using {format}]....]
     [under {name}] end
```

USAGE

Use the list command to output a list of fields or calculations performed on selected fields and to assign column headings to the fields or calculations displayed. You can enter any number of list commands. The order of results is determined by previous or subsequent "sort" commands.

If a name has been previously defined for an expression in a sort command, it may not be redefined. Otherwise, the name assigns a column heading to the expression. A "/" appearing in the name indicates that the heading continues onto a new line. You can use any number of "/"s in a name, indicating a multi-line stacked heading. Expressions, consisting of a single field, use the field name as their default heading. Otherwise, there is no default heading. After a heading has been established for an expression, use the heading in subsequent commands instead of the expression.

By default, expressions are assigned columns from left to right on line one. Either the output size of the expression or the heading size, whichever is larger, is used to determine spacing. You can alter default spacing in one of two ways. First, the line and column may be absolutely set after the "list" command. Second, you can use the "under" syntax to indicate that the expressions are to appear underneath a previously defined expression.

Numeric, float, and amount fields are normally printed using a default format. You can change the default with the "using" syntax and a "template" composed of special characters, one for

each position in the field width. The special character at each position in the template indicates what will be printed at that position. The special characters for numeric and amount fields are shown below:

#	If there is a digit in this position, print the digit. Otherwise, print a blank. This character pads a numeric field with blanks on the left.
&	If there is a digit in this position, print the digit. Otherwise, print a zero. This character pads a numeric field with zeros on the left.
*	If there is a digit in this position, print the digit. Otherwise, print an asterisk. You can use this character to print amounts on checks, for example, which are frequently padded with asterisks on the left.
$	If there is a digit in this position, print the digit. Otherwise, print a dollar sign. If a dollar sign has already been printed, print a space. Use this character to print either a fixed or floating dollar sign.
+	If there is a digit in this position, print the digit. Otherwise, print a plus sign. If a plus sign has already been printed, print a space.
-	If there is a digit in this position, print the digit. Otherwise, if this is a negative number, print a minus sign. If a minus sign has already been printed, print a space.
(	If there is a digit in this position, print the digit. Otherwise, if this is a negative number, print a left parenthesis. If a left parenthesis has already been printed, print a space.
)	If this is a negative number, print a right parenthesis in this position.
,	If there is a digit to the left of this position, print a comma. Otherwise, print a space.

Print a decimal point in this position.

For float type fields, use a print specification exactly like the "printf" function utilized by the C programming language. The print specification has the following format:

%[-][minimum field width][.][precision]f | e | g

The fields enclosed in square brackets ([]) are optional. The vertical bar (|) separating the list of items indicates that you must choose one of the items in the list. The percent sign (%) is required.

The optional minus sign (-) in front of the conversion specification indicates that the result is to be left justified in the field width. The minimum field width is a number which indicates the minimum number of print positions in the result. If the result has fewer characters, it is padded to fill the field width. Precision indicates the number of digits to appear after the decimal point. If precision is Ø, no digits are printed after the decimal point. If there is no dot, the number in front of the conversion character is assumed to be the precision. The conversion characters (f, e, and g) are defined below:

f The field is converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output in the format shown above; if the precision is explicitly Ø, then no decimal point is printed.

e The field is converted in the style "[-]d.ddde±dd", where there is one digit before the decimal point and the number of digits after the decimal point is equal to the precision specification. If the precision is explicitly Ø, then no decimal point is printed.

g The field is converted in style f or style e, whichever gives full precision in minimum space.

The following examples illustrate how various templates affect output:

Format	Value	Result
"#####"	123	" 123"
"#####.##"	Ø	" .ØØ"
"#####.&&"	Ø	" .ØØ"
"+++,+++,+++"	23456	" +23,456"
"---,---.&&"	23456.78	" 23,456.78"
"---,---.&&"	-2345.67	" -2,345.67"
"($$,$$&.&&)"	2345.ØØ	" $2,345.ØØ"
"($$,$$&.&&)"	-2345.67	"($2,345.67)"
"$##,##&.&&"	1234	"$ 1,234.ØØ"
"$**,***.&&"	123	" $**123.ØØ"
"%1Ø.2f"	12.3	" 12.3Ø"
"%1Ø.2f"	123.456	" 123.46"
"%12.4e	123.456	" 1.235 e+ØØ2"
"%1Ø.4g	123.456	" 123.4"
"%8.4g	123456789	"1.23e+Ø8"

SORT SORT

SYNTAX

```
sort [reverse]
     [uniquely]
     [{name}] {expr} [, [{name}] {expr}....] end
```

USAGE

Use this command to determine the listing output sequence. If the sort command is not used, the output appears in no particular sequence. By default, sort causes the output to be in ascending sequence by the specified expressions. If the reverse syntax is used, the listing sequence is reversed. The expressions are listed in order of decreasing sort importance. The optional name argument gives the expression a name which can be used in subsequent "total" and "list" commands. If a name is specified and the expression is listed or totaled, the name is used as a column heading.

The "uniquely" syntax specifies that identical lines are to be output only once. Note that the entire line must be identical to be suppressed, not just the sort expression.

TOTAL TOTAL

SYNTAX

```
total  |  {name}  |  [using {format}],....
       |  {expr}  |
                     [by {name} [, {name}]] end
```

USAGE

Use this command to output expression totals and subtotals. By default, the named expressions are totaled for the entire set of selected records. The "by" syntax allows totals to be produced at level breaks. The names specified after "by" are the names of expressions which have previously appeared in sort statements. A total is produced for each change in value for these expressions.

If the expression has been previously named, use only the expression name. If the expression is named in a list statement, the output column and format is the same as that specified in the list statement. If the expression does not appear in a list statement, you can use the name to give the expression a column heading. A "/" in the name indicates that the heading goes on a new line. You can use any number of "/"s in a name. If no name is specified and the expression consists of a single field, the heading is, by default, the field name. Otherwise, there is no default heading.

If the expression is not named in a previous list statement, you can use the using syntax to define the output format.

GO GO

SYNTAX

go

USAGE

This command causes the specified listing to be written to the standard output, which is, by default, the screen of the terminal from which the listing processor is executed. In interactive mode, the output is stopped every 23 lines to allow for user interaction. The column headings are printed unless the nohead command has been previously issued.

PRINT PRINT

SYNTAX

print {report-heading}

USAGE

This command outputs the specified listing to the print spooler (lpr). Page numbers, date, time, and report title are output, in addition to the specified column headings. The report-heading is output on every page. A "/" in the report-heading indicates a new line. You can use any number of "/"s in the report-heading to indicate a stacked, multi-line heading.

NOHEAD NOHEAD

SYNTAX

nohead

USAGE

When this command is issued prior to a "go" command, the output does not contain column headings. This command is useful for dumping data from the database to TRS-XENIX files without extraneous information.

UNLOCK UNLOCK

SYNTAX

unlock {field} {password}

USAGE

Use this command to unlock a field which is protected by field level security. You can use either the read or write password to unlock a field. If an attempt is made to access a secure field in a report without using this command, an error message is displayed.

DATABASE TEST DRIVER

DISREGARD THIS CHAPTER UNLESS OPERATING WITH THE TRS-XENIX DEVELOPMENT SYSTEM.

This program uses the record and explicit relationship functions described in the next chapter, "C Language Interface," in an interactive mode once you create a database file. Test data can be entered quickly and easily, and results of test runs can be checked. The syntax used is very simple. When the program is ready for input, the colon (:) appears on the terminal as a prompt. The function names and the record and field names appear as you entered them in the schema. If a function requires additional data, the program prompts for it. Validity checking is performed to make sure that correct names are used. If a function fails, the value of the return status code is printed out. The interpretation of these values is discussed in the next chapter. To exit the test driver, respond to the prompt with either "end" or "q."

The arguments to the functions which may be used in the Database Test Driver are record for record, name for field name, and rfield for the name of a reference field in another record. For more information on these functions, refer to the corresponding functions in the section on "C Language Interface."

access <record>
Enter the record key, and the indicated record appears.

addrec <record>
Enter the record key, and the indicated record is added to the file. Enter each field in the record, and when you have input all fields, the record is displayed.

clr <field> <rfield>
The indicated set of related records is cleared. This function corresponds to the clrset function.

delete <record>
The current record of the specified type is deleted.

faccess <record> <field>
The indicated record is accessed using the value of the specified field. (Note that a record of the type containing the field must be current.)

loc <record>
The location of the current record of the specified type is printed. This location can be used later with the setloc function.

makeset <field> <rfield> <option>
The indicated set of related records is identified for subsequent calls to nextrec or prevrec. The option is chosen from the set {first, last}.

next <field> <rfield>
The next record in the specified set of related records is made current. This function corresponds to the nextrec function.

<field>
By simply typing the name of a field, you can change its contents. This function corresponds to the pfield program function. The program prompts for field data and makes the indicated update.

prev <field> <rfield>
The previous record in the specified set of related records is made current. This function corresponds to the prevrec function.

samerec <field> <rfield>
The current record of the specified set of related records is restored.

seq <record> <option>
The indicated record is updated and displayed. Choose an option from the set {first, next, last, prev}. This function corresponds to the seqacc function.

setloc <record> <address>
The record at the specified location is updated. Invalid

record addresses are not allowed.

setsize <field> <rfield>
The number of records in the specified set of related records is displayed.

C LANGUAGE INTERFACE

DISREGARD THIS CHAPTER UNLESS OPERATING WITH THE TRS-XENIX DEVELOPMENT SYSTEM.

This section describes the various database manipulation functions available from the C programming language. Although C is the most common language on TRS-XENIX, Unify also interfaces with other languages, such as RM COBOL. These functions form the interface between user programs, CRT terminals, the printer, and the database. (You should already understand the concepts of a schema, record relationships, and the procedural manipulation of records in a database.) The Unify routines allow user programs to perform the following:

1. Add and delete records.
2. Change the contents of data fields.
3. Accept input from the screen and display output on the screen.
4. Move throughout the database using the different access methods available.
5. Format output to the printer.
6. Sort and select database records.

In order to use these functions, it is necessary to have already established a database file (refer to Chapter 2). Once you have created a database, you will have a file.h. File.h is a text file, which assigns UNITRIEVE internal numbers to each of the record and field names. This text file must be "included" with each user function which uses either field or record names.

In addition, it is necessary to include the UNITRIEVE database archive whenever loading programs that use database functions. This archive is named libd.a, and it is located in the unfiy/lib directory. If you have set the environment variable Unify to point to this directory, you can reference it as $Unify/libd.a. The utility functions, described later in this chapter, are kept in the archive libx.a, which is also located in the unify/lib directory. Reference it using $Unify/libx.a. A general purpose loading shell command file named uld is included as part of the Unify system in the unify/bin directory. Its usage

is described in the next section, "Compiling and Loading Programs."

In the following function descriptions, some conventions are used to name the arguments. The following is a brief description of these conventions.:

```
rnum   - Any record name
field  - Any field name
rfield - Any field that refers to the key field of
         another record (explicit relationship)
```

It is important to understand the difference between "internal" and "external" field formats. To save space and simplify manipulation, UNITRIEVE uses compacted forms in both the database and the user functions. These formats are expanded to external form when a field is sent to a display device, such as the screen or the printer. The schema describes the external form, but the internal form can be predicted using the following chart:

SCHEMA		INTERNAL	EXTERNAL
NUMERIC	(1-4)	short	9999
	(5-9)	long	999999999
FLOAT		double	variable
STRING		char[?]	xxxxxxx...
AMOUNT	(1-7)	long	9999991Ø.99
	(8-12)	double	9999999999991Ø.99
DATE		short (julian)	99/99/99
TIME		short	HH:MM

Only date fields are initialized to a null value other than Ø. All other fields are initialized to Ø. Null for date fields is represented by -32768 internally and **/**/** externally. You can set an amount field to null (which is different from Ø) by storing -32768 in the high order word and Ø in the low order word. Amount fields set to null are displayed as blanks externally.

NUMERIC, FLOAT, and STRING data types are stored exactly as their types imply. AMOUNT fields are stored internally as cents,

rather than as dollars and a fraction. The output routines use the MOD and DIV operators to position the decimal point. TIME fields are designed to store "time of day" information, rather than lengths of time, although lengths of up to 23 hours 59 minutes can be represented. The time "12:59" is stored internally as the integer 1259. The output routines use MOD and DIV, as with AMOUNT fields, to break the integer apart for display.

The UNITRIEVE functions use the notion of a "current" record. For each record type, one occurrence of the record can be current. This current record is the implied argument to many functions. When the program begins, all record types are non-current. Before accessing a record, you must first make it current by using addrec, access, faccess, nextrec, prevrec, samerec, nextsel, prevsel, and so on. The record then remains current until another record of the same type replaces it. UNITRIEVE faults a program which attempts a function for which there is no current record.

Compiling and Loading Programs

This section describes how to compile and load programs when using the Unify host language interface with C programs. In general, the process is like compiling and loading any C program, except a special preprocessor is used, and you must include the Unify libraries in the load. The following two function descriptions explain how to use ucc, the Unify C compiler, and uld, the Unify load macro.

UCC UCC

NAME

ucc -- Unify C compiler preprocessor

SYNTAX

ucc -c [-O] file...

DESCRIPTION

Ucc is a program which compiles Unify application programs written in C. Ucc calls a preprocessor to manipulate the include file file.h. The preprocessor is needed, because the preprocessor which comes with TRS-XENIX cannot handle "define" names longer than eight characters. Unify-generated field names can be much longer than eight characters and are present in file.h. The Unify preprocessor looks for $include rather than #include, so an application program which uses Unify field names must contain a line similar to the following:

$include "../../def/file.h"

After running the Unify preprocessor, ucc calls cc (C-compiler) with the same arguments passed to ucc. The ucc command is shown below:

ucc -c [-O] filel.c ... filen.c

The -O is optional. This option invokes the optimizer for the C-compiler. If the various files compile successfully, the objects are left in .o files. Currently, it is not possible, using ucc, to produce an executable object directly by omitting the -c. The object files created should be loaded using uld, the Unify load shell file.

EXIT STATUS CODES

Ø -- program compiles successfully
1 -- preprocess step failed
2 -- compile failed
3 -- interrupt terminated the compile

SEE ALSO

uld

BUGS

Single quotes appearing in a comment can cause preprocessing to terminate, leading to undefined database fields.

Example: /* set today's date */

This type of comment causes problems, so leave it out.

ULD ULD

NAME

uld -- Unify load macro

SYNTAX

uld file...

DESCRIPTION

The command file uld is a general purpose macro for loading Unify programs. It references **libd.a** and **libx.a**, which are archives containing the Unify host language functions. This command file uses the environment variable $UNIFY to locate the Unify library archives, so you must set this variable or modify the uld script before using the command file. The C extension and math libraries for TRS-XENIX named libc.a and libm.a are also referenced by the Unify load macro.

The uld (uuld) command is shown below:

uld binary [filel.o ... filen.o] [filel.a ... filen.a]

The argument **binary** is the name of the executable to be created as the output from the load. The remaining arguments are the necessary **.o** and **.a** files needed to create the executable. No more than eight .o and .a files can be passed as arguments. (Refer to Chapter 2 for more information on the use of uld and the resulting executable.)

When using this function, conflicts may result between TRS-XENIX and Unify symbol names. These problems appear as warning messages about multiply defined symbols. In general, you can ignore these messages, because the Unify routine is the correct one to load.

Unify Error Handling

Errors from C programs are divided into two major classes -- fatal errors and non-fatal errors. Fatal errors display a message to identify the problem and then abort the program in question. You can then use adb to examine the resulting core dump. An example of a fatal error is failing to successfully open the database file (file.db) for reading and writing. Fatal errors never happen in a normally functioning system. Their occurrence indicates something is seriously wrong with either the program or file.db.

Non-fatal errors simply return a status code to the program. An example of a non-fatal error is failing to access a record with a particular key value. Note, however, that non-fatal errors, if ignored, can lead to fatal errors. For example, attempting to get a field from a record without checking the status from a preceding access or nextrec can yield a fatal error, if the access (nextrec) failed.

The following section describes fatal errors only, since non-fatal errors are documented in the section for each particular function. The possible fatal errors are not listed with each function, because a number of errors can occur for most of the functions. In addition to the possible fatal errors, any function that takes record and/or field numbers as arguments can have a completely unpredictable result if an invalid number is passed. This problem is most likely to occur as a result of passing a field number instead of a record number. Strange problems (memory fault, bus error, or unexpected fatal errors) are sometimes caused by this kind of an error.

There are three general types of fatal errors: user errors -- the user program is attempting an illegal or undefined operation; internal errors -- problem with a Unify function; and database errors -- problem with file.db. It is not always possible for the function to distinguish the type of error, so the programmer must determine the problem.

The following rules are used to differentiate between the various types of errors. If the program is under development and

uses a small, non-volatile test database, a fatal error is almost always a program bug. Look for improper use of functions, failure to take appropriate action on non-fatal errors, or programs overwriting either their data or text space (in that order). A program bug is very likely if a particular program is the only one having problems and other debugged programs work properly.

In a well-debugged production program, the possibility still exists for an undiscovered bug. If the program does not appear to have a bug, the next most likely possibility is corruption of the database file, file.db. This cause is possible if either a fatal program error or hardware failure occurs in the middle of a transaction. If other programs are having similar problems, the problem almost certainly lies with the database file itself.

The data dictionary and/or header information can be corrupted, the hash index can be invalid, or some number of pointers can be bad. The fastest and easiest fix is to read in the latest backup of file.db and start over. If this is not possible or desirable, there are various means of fixing the file. You can rebuild the data dictionary and header information by running Reconfigure Database (Chapter 3). You can rebuild the hash index by executing Hash Table Maintenance, and fix the pointers by using Explicit Relationship Maintenance. If rebuilding the file does not fix the problem, you can dump the data to ASCII files using SQL (Chapter 6), create the database again, and then reload the data using Database Load.

If you want to catch fatal errors and do something other than abort, you can load your own error handling routine ahead of the ones supplied with Unify. Unify provides four different routines to trap errors. Their names and calling sequences are listed on the next page. If you want to use a different routine, simply load your .o explicitly ahead of the Unify archives libd.a and libx.a, and your routine is used instead.

ERROR ERROR

NAME

error, uerror, dbherr, pferr -- trap fatal errors

SYNTAX

```
error (str, ier)
char *str;

uerror (str, ier)
char *str;

dbherr (str)
char *str;

pferr (ier, str)
char *str;
```

DESCRIPTION

Unify uses these routines to trap fatal errors. The indicated message is printed, and exit is called, with status code 99. Error is used mainly to report database internal errors; uerror reports user errors; dbherr reports internal errors dealing with the data dictionary; and pferr reports user errors dealing with incorrect usage of pform.

If you wish to perform some other type of error handling, write C routines with the same names and calling sequences and load .o ahead of the Unify archives, libd.a and libx.a.

RETURNS

A "99" status is returned to the shell.

FATAL ERRORS

*** /bin/sort not found

Explanation:
The TRS-XENIX sort utility, /bin/sort, cannot be executed.
Possible Causes and/or Solutions
1. /bin/sort may not exist. Look in unify/bin if sort is not in /bin. If this is the case, copy the file to /bin.
2. The current TRS-XENIX user cannot execute the program. It may be necessary to change the permission of directories in the path name as well as the sort program itself.

*** can't open xxxxxxxx
database error
from loadscr ier= 1

Explanation:
User error. The indicated screen .q file cannot be opened for reading.
Possible Causes and/or Solutions
1. If the file doesn't exist in the current directory, create it or recover it from diskette.
2. If the file does exist, change the permission so it is readable for the current TRS-XENIX user, or log on as a user with reading privileges.

*** database error
from gett ier= n

Explanation:
User error, internal error or database error. There are four different types of errors diagnosed, depending on the value of n.
Possible Causes and/or Solutions
1. -1 means an invalid record number was passed to gett

(less than 1 or greater than the max record number). The most common user error that causes this problem is passing a field number where a record number is expected, for example, trying to "access" a record with a field number. If the function parameters are correct, an internal error occurs. Possible causes are the program overwriting itself, a Unify routine overwriting itself, or file.db is bad.

2. -2 means the offset within the record from which the data comes is less than 1. The same comments regarding internal errors apply as in 1. above.
3. -3 means the number of words to read is less than 1 or greater than the record length. The same comments regarding internal errors apply as in 1. above.
4. -4 means there is no current record of the type that contains the specified field. Usually, this problem is a user error caused when a function, such as access, nextrec, or prevrec, fails, and the program continues on blindly. In rare cases, this problem can be a database error.

*** database error
from input ier= 1

Explanation:
User error. An attempt has been made to input (store in the database) a screen field for which there is no database field defined.

Possible Causes and/or Solutions
1. Define a database field for the screen field and recompile the screen.
2. Otherwise, use "inbuf" to return the operator response to a program buffer.

*** database error
from lifo ier= -1

Explanation:

Internal error. This error occurs only if internal consistency checks fail.

Possible Causes and/or Solutions

Report the circumstances of the error to your Radio Shack Computer Representative.

*** database error
from put ier= n

Explanation:

User error, internal error or database error. There are four different types of errors diagnosed, depending on the value of n.

Possible Causes and/or Solutions

1. -1 means an invalid record number was passed to put (less than 1 or greater than the max record number).
2. -2 means the offset within the record, from which the data comes, is less than 1. The same comments apply as in 1. above.
3. -3 means the number of words to write is less than 1 or greater than the record length. The same comments apply as in 1. above.
4. -4 means there is no current record of the type that contains the specified field. Usually, this problem is a user error, when a function, such as access, nextrec, or prevrec fails, and the program continues on blindly. In rare cases, this problem can be a database error.

*** db:nnn addr:nnn errno:nnn
database error
from put-dwrit ier= -1

Explanation:

Internal error or database error. This problem is an error from a database write attempt. Db is the write file descriptor, addr is the byte address of the write, and errno is the TRS-XENIX error number returned from the operation.

Possible Causes and/or Solutions

1. File.db does not exist, or the current TRS-XENIX user does not have writing privileges to the file.
2. User trying to write to an invalid starting address or trying to write an invalid number of bytes. Either the program is overwriting itself or the file.db is bad and needs to be recovered.

```
*** database error
      from recphys ier= -2

Explanation:
     Database error. This problem is a serious error in the
     header information of file.db.
Possible Causes and/or Solutions
     Recover the database file by running Reconfigure Database.


*** database error
      from rloc ier= -1

Explanation:
     Internal error or database error. This problem is a serious
     error in either a Unify routine or the file.db itself.
Possible Causes and/or Solutions
     Recover the database by running Reconfigure Database.


*** database error
      from unlk ier= -1

Explanation
     Internal error. This error occurs only if internal
     consistency checks fail.
Possible Causes and/or Solutions
     Report the circumstances of the error to your Radio Shack
     Computer Representative.
*** error from dread nnn
     addr nnn
```

```
     database error
      from dread-gett ier= -1

Explanation:
     Internal error or database error. This problem is an error
     from a database read attempt. The error code from dread
     (nnn) is the return value from the read attempt, while addr
     is the starting read address, in words.
Possible Causes and/or Solutions
     1. File.db does not exist or is not readable by the current
        TRS-XENIX user.
     2. User trying to read from an invalid starting address or
        trying to read an invalid number of bytes. Either the
        program is overwriting itself or the file.db is bad and
        needs to be recovered.


*** HTLEND nnn HTLSTRT nnn lloc nnn
     database error
      from gthsh ier= -1

Explanation:
     Internal error or database error. This error is essentially
     the same as the previous error.
Possible Causes and/or Solutions
     Follow the steps shown above.


*** HTLEND nnn HTLSTRT nnn lloc nnn
     database error
      from gthsh-hcheck ier= -1

Explanation:
     Internal error or database error. This problem is a serious
     program or database error. A hash table location falls
     outside the bounds of the hash table. HTLEND is the hash
     table end address, HTLSTRT is the start address, and lloc
     is the location that caused the fault.
Possible Causes and/or Solutions
     1. The header information in file.db is corrupt. If all
```

other programs have the same problem, this is almost certainly the case. Recover the file by running a reconfiguration.
2. If other programs don't have the problem, it is a user error or an internal error. Check to make sure the program is not writing over itself.

*** illegal field -gtube- nnn
database error
from gtube ier= nnn

Explanation:
User error. Gtube receives an unrecognized field number. The error code (ier) represents the field number passed to gtube.

Possible Causes and/or Solutions
1. The database field may have been deleted. If so, the program must be redesigned.
2. An incorrect or out-of-date file.h may have been included by the program. Find the correct file.h and recompile and reload the program.
3. If an integer variable is used for the field number instead of a define from file.h, check to make sure it is being used correctly.

*** insufficient memory for data dictionary

Explanation:
User error. There is not enough room in memory for the data dictionary information.

Possible Causes and/or Solutions
1. Load the program using the -i option of the loader, if it is not already loaded this way. This allows for 64K of data and 64K of program text.
2. Decrease the amount of space allocated for global variables. Either remove some data and programs from the core load or move some global data to the stack by making it local.

*** insufficient memory for additional dictionary information

Explanation:
User error. This error is basically the same problem shown above.
Possible Causes and/or Solutions
Follow the same procedures recommended above.

*** NO MORE SPACE FOR RECORD nnn
database error
from fetchbuff ier= -1

Explanation:
User error. There is no more space in file.db to store additional records of type nnn. This error occurs when the file has 66% more records than predicted in the schema design.
Possible Causes and/or Solutions
1. Delete some existing records of type nnn. This solution frees space to add more records.
2. Use Schema Maintenance to increase the expected number of records of the indicated type, and then reconfigure the database.

*** Pform error no 1 from getpf

Explanation:
User error. Insufficient arguments or an argument not separated by commas on the current line in the pform input file.
Possible Causes and/or Solutions
Find the incorrect line in the input file and correct it.

*** Pform error no 2 from getpf

Explanation:
User error. The conversion of an x coordinate from ASCII to decimal failed.
Possible Causes and/or Solutions
Examine the current pform input file for non-numeric characters in the x coordinate position.

*** Pform error no 3 from getpf

Explanation:
User error. Conversion of a y coordinate from ASCII to decimal failed.
Possible Causes and/or Solutions
Examine the current pform input file for non-numeric characters in the y coordinate position.

*** Pform error no 4 from getpf

Explanation:
Internal error.
Possible Causes and/or Solutions
This error should not occur. If it does, report the circumstances to your Radio Shack Computer Representative.

*** Pform error no 5 from getpf

Explanation:
User error. An x or y coordinate exceeds three numeric characters in length.
Possible Causes and/or Solutions
Examine the current pform input file and correct the offending line.

*** Pform error no 6 from getpf

Explanation:
User error. The user-supplied function name is longer than 1Ø characters.
Possible Causes and/or Solutions
Examine the current pform input file for the long name. Shorten the name to 1Ø characters or less. This change may require that you rename the function and change all calls to the new name.

*** Too many unisort sets are current

Explanation:
User error. Only eight different unisort sets can be current at any given time.
Possible Causes and/or Solutions
Check the program to determine if too many different unisort sets are being used at the same time. In particular, make sure that you are using clrset where appropriate.

*** unable to open database file

Explanation:
User error. File.db in the current directory cannot be opened for update.
Possible Causes and/or Solutions
1. File.db does not exist in the current directory.
2. The current TRS-XENIX user does not have read/write access to file.db. Either change the mode of the file or log on to TRS-XENIX with a valid read/write code.

*** unexpected EOF on FN file

Explanation:
Database error or internal error. This error indicates a serious problem exists with the data dictionary information at the beginning of file.db.
Possible Causes and/or Solutions
Recover file.db by running Reconfigure Database.

*** user error
illegal field number *gfield*
error code nnn

Explanation:
User error. The gfield function receives an invalid field number. The error code (nnn) represents the invalid field number passed to gfield.
Possible Causes and/or Solutions
1. An incorrect or old file.h may be causing the problem. Make sure the file.h you are including with the program is correct.
2. The integer variable you are passing as the argument may contain an out-of-bounds value.
3. The program may be overwriting itself.

*** user error
illegal field number *pfield*
error code nnn

Explanation:
User error. The pfield function receives an invalid field number. The error code (nnn) represents the invalid field number passed to pfield.
Possible Causes and/or Solutions
1. An incorrect or old file.h may be causing the problem. Make sure the file.h you are including with the program is correct.
2. The integer variable you are passing as the argument may

contain an out-of-bounds value.
3. The program may be overwriting itself.

Record Functions

This section describes the record-oriented UNITRIEVE database functions. These functions allow user programs to manipulate records and fields within the records according to the schema description. These functions are low-level, one record at a time functions.

ACCESS ACCESS

NAME

access -- access a database record

SYNTAX

```
access(rnum,key)
char *key;
```

DESCRIPTION

Access finds a record in the database by its key. Rnum specifies the type of record to find. Key is a pointer to a buffer containing the record key. Since it is assumed that the buffer is the length of the key, no null terminator is needed.

RETURNS

```
 Ø -- Record located and is now current
-1 -- No key for this record type
-2 -- No record of this type with this key
```

When accessing a record with a combination field, use a structure that has the same format as the internal format of the key field. Since all fields start on word boundaries, odd length character fields must be padded. Even if the odd length character field occurs as the last field of the combination, it still must be padded. You must use the same padding character when accessing the record you used when adding it, or the record will not be found. ENTER and the database test driver use a null for padding.

The name of this function conflicts with the TRS-XENIX system call access(2). Use the system call stat(2), if you want to see if a file is accessible.

ACCSFLD ACCSFLD

NAME

accsfld -- make a protected database field accessible

SYNTAX

```
accsfld(fnum,pass,priv)
int fnum;
char *pass,priv;
```

DESCRIPTION

Use this function to allow a program to access a database field protected with Field Level Security. A password is needed to unlock the field. If the password is correct, all fields in the same group are opened to access. The user can specify whether read or read and write is desired. The choice may require use of a different password.

ARGUMENTS

fnum -- the field number
pass -- pointer to a null terminated password
priv -- 'r' for read only, 'w' for read/write

RETURNS

Ø -- field is now accessible
-1 -- invalid password

ADDREC **ADDREC**

NAME

addrec -- add a new record into the database

SYNTAX

```
addrec(rnum,key)
char *key;
```

or

```
addrec(rnum)
```

DESCRIPTION

Addrec inserts a new record into the database with the given unique key. If the key is unique, the record is added and is made current. If the key is not unique, an error is returned, and the old record with this key is made current. If the record type has no key, the second argument is not needed, and the record is always inserted.

RETURNS

```
 Ø -- the record has been added and is now current
-1 -- the key is not unique and the old record is
      current
-2 -- the key refers to a non-existing record. This is
      the same as a -3 from pfield
-3 -- write access is not allowed on some field in the
      record (see accsfld)
```

When adding a record with a combination field, use a structure that has the same format as the internal format of the key field. Since all fields start on word boundaries, odd length character fields must be padded. Even if the odd length character field occurs as the last field of the combination, it still must be padded. You must use the same padding character when adding

the record as you used when accessing it, or the record will not be found. ENTER and the database test driver use a null for padding.

CLOSEDB CLOSEDB

NAME

closedb -- close a database file

SYNTAX

closedb ()

DESCRIPTION

This function closes a database file opened by opendb. It does not deallocate the memory space grabbed by opendb.

RETURNS

none

DELETE DELETE

NAME

delete -- delete a record from the database

SYNTAX

```
delete(rnum)
```

DESCRIPTION

This function removes a record from the database. The record to be deleted must be current. The current record is removed and is not accessible to further access, nextrec, prevrec, or samerec function calls. If the record has any existing related records, using this function is not legal. All of the related records must be deleted before the record can be deleted.

After a successful delete, the current record is unpredictable. After an unsuccessful delete, the current record remains the same. Note that when records are deleted, the space is available for later inserts of the same record type.

RETURNS

```
 Ø -- record is deleted
-1 -- record not deleted
-2 -- write access not allowed on some field in the
      record (refer to accsfld)
-3 -- record is locked by another process (see
      lockrec)
```

ENDTRANS **ENDTRANS**

NAME

endtrans -- end transaction

SYNTAX

endtrans ()

DESCRIPTION

Use this function to unlock the database at the end of an update transaction. This function is the inverse of startrans.

Two different implementation methods are used depending on if a TRS-XENIX system call that supports file locking is available. If a locking system call is available, byte Ø of the file lockfile (ddlockfile, if unify.db is being unlocked) is unlocked using the system call. If the locking system call is not available, the Unify function unlock is used instead.

The locking files are created in the user's working directory. Normally, this is the bin directory, but if $DBPATH is set, the working directory is the directory to which $DBPATH points. Note that this implies that each application must have a single working directory that everyone uses -- either the bin directory for the application or the same $DBPATH for everyone. If this rule is not followed, concurrency control is defeated, and the database integrity is destroyed.

RETURNS

none

SEE ALSO

startrans, unlock

FLDESC **FLDESC**

NAME

fldesc -- database field description

SYNTAX

```
#include "dbtypes.h"
#include "fdesc.h"
fldesc (field, fdsc)
int  field
FLDESC *fdsc;
```

The following is a description of FLDESC:

```
#define FLDESC struct fdesc
struct fdesc {
     int f_rec;          /* record */
     int f_typ;          /* field type */
     int f_len;          /* display length */
     int f_par;          /* comb field if a subfield */
     int f_rprec;        /* referenced record if a relation */
     int f_rpfld,        /* referenced field if a relation */
};
```

The possible field types defined in dbtypes.h:

INT, LONG, DATE, AMT, STRING, HAMT, HR, FLT, COMB

DESCRIPTION

This function returns the attributes of a database field in the structure pointer fdsc. If file.h is included in the program, the field should **not** be declared as an integer.

ARGUMENTS

field -- the database field
fdsc -- the address of a FLDESC structure

RETURNS

Ø -- invalid field
1 -- normal return

SEE ALSO

sfldesc

GFIELD GFIELD

NAME

gfield -- move a field from the database to a buffer

SYNTAX

```
gfield(field, buf)
char *buf;
```

DESCRIPTION

This function retrieves a field from the database and stores it in a buffer. The record that contains the field must be current. The field is still in internal format in the buffer. The field is retrieved from the current record of the type containing the field. The field must exist in file.h; otherwise, an error message concerning the illegal field number *gfield* is issued.

RETURNS

Ø -- the field was retrieved
-1 -- read access not allowed on this field (see accsfld)

LOC LOC

NAME

loc -- obtain record location

SYNTAX

```
loc(rnum,addr)
long *addr;
```

DESCRIPTION

This function returns the address of the current record of type rnum. If no record is current, the address returned is Ø.

RETURNS

none

LOCKREC LOCKREC

NAME

lockrec -- lock the current record

SYNTAX

lockrec (rnum)

DESCRIPTION

This function locks the current record of the specified type. Attempts to update a locked record using either **pfield** or **delete** will fail. Two different implementation methods are used depending on whether a TRS-XENIX system call that supports file locking is available. If a locking system call is available, use it to lock the appropriate bytes of the file lockfile (ddlockfile, if records in unify.db are being locked). If the locking system call is not available, the Unify function lock is used to protect against concurrent access, and an entry is made in lockfile (ddlockfile) that tells which record is locked.

The locking files are created in the user's working directory, which is normally the bin directory but can be in the directory $DBPATH. Note that this implies that each application must have a single working directory that everyone uses -- either the bin directory for the application or the same $DBPATH. If this rule is not followed, concurrency control is defeated, and database integrity is destroyed.

Note that since UNITRIEVE uses field level I/O (as opposed to record level I/O), it is generally not necessary to lock a record while updating it. Another process updating different fields in the same record does not write over fields your process is changing.

A process must unlock all locked records before it terminates. If a process that has locked records is killed or

dies and the locking system call is not available, the records remain locked. The only way to release the locked records is by removing the locking file (either lockfile or ddlockfile). You can remove these files safely only when no one is using the application. Removing either file while people are using the application destroys database integrity.

RETURNS

Ø -- record locked
-1 -- some other process has the record locked

OPENDB OPENDB

NAME

opendb -- open a database file

SYNTAX

```
opendb (dbname, opt)
char *dbname;
int opt;
```

DESCRIPTION

Use this function to allow a program to open database files, other than the default file, file.db. The user-level host language functions open file.db, if there is no other database file open. To open a different database file, call opendb before any other Unify function. Reading in the names and/or synonyms can take up considerable space, so you must have enough data space in your program. Opendb faults if available memory space is exceeded.

ARGUMENTS

dbname -- pointer to the database file name
opt -- tells opendb what additional dictionary information, if any, to read. The option is constructed by OR'ing together some combination of the following identifiers, which are declared in file.h. If you don't want to read in any names, pass literal Ø.
FN -- read field names
FS -- read field synonyms
RN -- read record names
RS -- read record synonyms

RETURNS

none

PFIELD PFIELD

NAME

pfield -- update a field in the database

SYNTAX

```
pfield(field,buf)
char *buf;
```

DESCRIPTION

Pfield updates the specified field with the contents of the buffer. The record that contains the field must be current. The data in the buffer should be in internal format. The data is stored into the current record of the type containing the field. Pfield can be used with any field, but some restrictions regarding key fields and relational fields should be noted. Performing a pfield into the key of a record changes the key, thus the data must define a unique key, just as in addrec. If there are related records that reference the key, using pfield is not allowed, as these records all contain fields that match the key being changed. (This change would lead to a logically inconsistent database.) The correct way to make this change is to pfield the relational field in each of the related records to the desired new value. Note that a record must exist with the new key value. These tests ensure that logical database integrity as defined in the schema is maintained.

RETURNS

```
 Ø -- the field is stored
-1 -- attempt to change the key of a record which
      currently has other records referencing it
-2 -- attempt to store a duplicate key
```

-3 -- attempt to form an explicit relationship to a record which does not exist
-4 -- write access not allowed (see accsfld)
-5 -- record is locked by another process (see lockrec)

SEQACC SEQACC

NAME

seqacc -- access all records in a file sequentially

SYNTAX

```
seqacc (rnum,direction)
```

DESCRIPTION

Use this function to retrieve all records in a file in the order in which they are stored, for example in logical record number order. This order is basically a random order, since space freed by deleted records is used to store new records added to the file. Rnum is the record type to retrieve, while option is chosen from the following set:

```
first -- access the first record in the file
next  -- access the next sequential record in the file
last  -- access the last record in the file
prev  -- access the previous sequential record in the
         file
```

RETURNS

```
 Ø -- the specified record is current
-1 -- specified record does not exist
```

SETLOC SETLOC

NAME

setloc -- set record location

SYNTAX

```
setloc(rnum,loc)
long loc;
```

DESCRIPTION

This function allows the user to set the current record for any given record type. The second argument is a record location, which is acquired using "loc." After the function call, the current record is the one at the location specified.

RETURNS

```
 Ø -- specified record is current
-1 -- invalid record type
-2 -- invalid record address
-3 -- record is deleted
```

STARTRANS STARTRANS

NAME

startrans -- start transaction

SYNTAX

startrans ()

DESCRIPTION

Use this function to lock the database at the beginning of an update transaction. The Unify functions addrec, delete, and pfield use startrans. All other database updates are suspended until the lock is released by calling endtrans.

Two different implementation methods are used depending on whether a TRS-XENIX system call that supports file locking is available. If a locking system call is available, byte Ø of the file lockfile (ddlockfile, if unify.db is being locked) is locked. If the locking system call is not available, the Unify function lock is used instead.

The locking files are created in the user's working directory. Normally, this working directory is the bin directory, but if $DBPATH is set, the working directory is the one to which $DBPATH points. Note that this implies that each application must have a single working directory that everyone uses -- either the bin directory for the application or the same $DBPATH for everyone. If this rule is not followed, concurrency control is defeated, and database integrity is destroyed.

RETURNS

none

SEE ALSO

endtrans, lock, lockrec, unlockrec

ULOCKREC ULOCKREC

NAME

ulockrec -- unlock the current record

SYNTAX

ulockrec (rnum)

DESCRIPTION

This function unlocks the current record of the specified type. This function is the inverse of lockrec.

Two different implementation methods are used depending on whether a TRS-XENIX system call that supports file locking is available. If a locking system call is available, use it to unlock the appropriate bytes of the file lockfile (ddlockfile, if records in unify.db are being unlocked).

If the locking system call is not available, the Unify function lock is used to protect against concurrent access, and the appropriate entry in lockfile (ddlockfile) is removed.

The locking files are created in the user's working directory. Normally, the working directory is the bin directory, but if $DBPATH is set, the working directory is the directory to which $DBPATH points. Note that this implies each application must have a single working directory that everyone uses -- either the bin directory for the application or the same $DBPATH for everyone. If this rule is not followed, concurrency control is defeated, and database integrity is destroyed.

A process must unlock all locked records before it terminates. If a process that has locked records is killed or dies and the locking system call is not available, the records remain locked. The only way to release the locked records is by removing the locking file (either lockfile or ddlockfile). You

can remove these files safely only when no one is using the application. Removing either file while people are using the application destroys database integrity.

RETURNS

none

SEE ALSO

lockrec, lock

Selection Processor Functions

This section describes the host language interface to create and manipulate query subset files (selection files) for use either by the LST or your own application programs. These functions let you create selection files or use existing selection files, created by the query processor, ENTER, or other programs, to perform additional processing. Records can either be returned in random order or sorted by multiple level sort keys as desired. The access method used is chosen by the functions, based on the selection parameters you specify. These functions form a high-level interface to the four different UNITRIEVE access methods (hashing, B-trees, explicit relationships, and buffered sequential access).

The package can be broken down into three major pieces. The highest level routines are unisel, ufsel, and selsort. These routines are the ones that either create a selection file (unisel), call a user function for each selected record instead of creating a selection file (ufsel), or create a list of sorted, selected records (selsort). A program typically uses one of these functions, depending on what it wants to do.

The second part of the package is the routines to set up the selection process by specifying the fields to be used in the selection and the criteria to use for selection. These routines are entsitm, mtchitm, fncitm, clrsitm, clrfitm, uqsrch, and uqmtch.

The third part of the package is a set of routines to manipulate the selection files themselves, once they have been created. These routines are opensf, closesf, frstsel, nextsel, and prevsel. Some programs may only use these functions to manipulate selection files created by the query processor, ENTER, or another program.

CLOSESF CLOSESF

NAME

closesf -- close selection file

SYNTAX

```
#include "unisel.h"
close (sf)
SELFILE *sf;
```

DESCRIPTION

Use this function to close a selection file which was opened using "opensf." Sf is the pointer returned from the opensf call.

RETURNS

none

SEE ALSO

opensf

CLRFITM **CLRFITM**

NAME

clrfitm -- clear function selection item

SYNTAX

```
clrfitm (sfnc)
int  (*sfnc) ();
```

DESCRIPTION

Use this function to clear a function selection item from the selection table used by unisel. Sfnc is the address of the function and is used to identify the item to be deleted.

RETURNS

none

SEE ALSO

sfncitm, clrsitm

CLRSITM **CLRSITM**

NAME

clrsitm -- clear selection item

SYNTAX

clrsitm (field,rfield)

DESCRIPTION

Use this function to clear a selection item from the selection table used by unisel. The arguments are used to identify which item is to be deleted, since the combination must be unique in the table.

ARGUMENTS

field -- the field upon which the selection is based
rfield -- the referenced field which is used to match items. Rfield should be literal Ø if there is no referenced field.

RETURNS

none

SEE ALSO

entsitm, mtchitm, unisel

ENTSITM ENTSITM

NAME

entsitm -- enter selection item

SYNTAX

```
#include "unisel.h"
entsitm (field, val1, val2, mode)
int mode;
char *val1, *val2;
```

DESCRIPTION

Use this function to enter a selection item into the selection table used by unisel. If the item is already in the table (field is unique), an attempt is made to merge the current item with the new item to create a range. If the attempt fails, the item is replaced.

ARGUMENTS

field -- the field upon which the selection is based

val1 -- the address of a buffer containing the value of the field used in the selection. The type of the value must agree with the internal field type.

val2 -- the address of a buffer containing the second value if the selection is a range. If the selection is not a range, set to Ø. Mode should be EQ or NOT for a range.

mode -- the mode of the selection: GT, GTE, LT, LTE, EQ, NOT. These modes are defined in "unisel.h."

RETURNS

```
 Ø -- normal termination
-1 -- the field is invalid
-2 -- the argument, vall, is not set
-3 -- the mode is illegal
-4 -- the selection table is full
```

SEE ALSO

mtchitm, sfncitm, clrsitm

FRSTSEL **FRSTSEL**

NAME

frstsel -- get first selected record

SYNTAX

```
#include "unisel.h"
frstsel (sf)
SELFILE *sf;
```

DESCRIPTION

Use this function to make the first selected record current. Sf is an address of a SELFILE structure, which was returned from a previous opensf call. To use this function, the file must be opened by opensf.

RETURNS

```
 1 -- the first selected record is current
-1 -- there are no selected records in the selection
      file
```

NOTES

Records retrieved using frstsel and nextsel are returned in logical record number order.

SEE ALSO

nextsel, prevsel, opensf

MTCHITM **MTCHITM**

NAME

mtchitm -- enter match item

SYNTAX

```
mtchitm (selfile, field, rfield)
char *selfile;
```

DESCRIPTION

Use this function to enter a match item into the selection table for unisel. A match item describes a relation between the current record type to be selected and an existing selection file.

ARGUMENTS

selfile -- a null terminated string containing the name of the related selection file.

field -- the field in the target record which references a field in the record type of the related selection file.

rfield -- the referenced field in the record type of the related selection file.

RETURNS

Ø -- normal termination
-1 -- the related selection file is not in the correct format
-2 -- the selection table is full

SEE ALSO

entsitm, sfncitm, clrsitm

NEXTSEL **NEXTSEL**

NAME

nextsel -- get next selected record

SYNTAX

```
#include "unisel.h"
nextsel (sf)
SELFILE *sf;
```

DESCRIPTION

Use this function to make the next selected record of a selection file current. Sf is an address of a SELFILE structure, which was returned from an opensf call. To use this function, the file must be opened by opensf.

RETURNS

```
 1 -- the next selected record is current
-1 -- the end of the selection file has been reached
```

SEE ALSO

frstsel, prevsel, opensf

OPENSF **OPENSF**

NAME

opensf -- open selection file

SYNTAX

```
#include "unisel.h"
SELFILE *opensf (selfile)
char *selfile;
```

DESCRIPTION

Use this function to open a selection file created by unisel, the query processor, or ENTER. The return from this call is an address of the type SELFILE and is used in subsequent calls to frstsel, prevsel, and closesf. The easiest way to use this function is shown below:

```
sample ( )
{
     SELFILE *sf, *opensf();

     sf = opensf ("selfile");
}
```

ARGUMENTS

selfile -- the address of a null terminated string containing the name of the selection file to open

RETURNS

-1 -- an error occurred and erno is set as follows:
- 101 -- selfile could not be opened
- 102 -- selfile is not in the correct format
- 103 -- the maximum number of open selection files has been reached
- 104 -- an error due to **sbrk**

The normal return is an address pointing to a SELFILE structure.

SEE ALSO

frstsel, nextsel, prevsel, closesf

PREVSEL PREVSEL

NAME

prevsel -- get previous selected record

SYNTAX

```
#include "unisel.h"
prevsel (sf)
SELFILE *sf;
```

DESCRIPTION

Use this function to make the previous selected record of a selection file current. Sf is an address of a SELFILE structure, which was returned from an opensf call. To use this function, the file must be opened by opensf.

RETURNS

```
 1 -- the previous selected record is current
-1 -- the beginning of the selection file has been
      reached
```

SEE ALSO

nextsel, frstsel, opensf

SELSORT SELSORT

NAME

selsort -- selection file sort

SYNTAX

```
#include "unisel.h"
selsort (rnum,fldtbl,srtfunc)
int *fldtbl;
int (*srtfunc[]) ();
```

DESCRIPTION

This function forms a shell around unisel that sorts the selection file that unisel creates, in much the same way that unisort sorts a set of records defined by a makeset call (see the next section, "Explicit Relationship Functions"). Before selsort is called, the selection table is created with calls to entsitm, mtchitm, and/or sfncitm.

When selsort is called, a selection file of record type rnum is created and then sorted based upon the values in fldtbl and srtfunc. (Refer to unisort for a description of these two arguments.) After the selection and sort, the records are retrieved using sfrstrec, snextrec, sprevrec, and slastrec. The arguments to use with these functions are: any field in record rnum, and SS (a define in unisel.h); instead of field and rfield.

RETURNS

-1 -- no records were selected by unisel
-2 -- the selection file created by unisel could not be opened
Ø -- normal termination

SEE ALSO

unisel, unisort, sfrstrec, snextrec, slastrec, sprevrec, entsitm, mtchitm, sfncitm

SFNCITM **SFNCITM**

NAME

sfncitm -- enter function selection item

SYNTAX

```
sfncitm (sfunc)
int (*sfunc) ();
```

DESCRIPTION

Use this function to enter a selection item into the unisel selection table, which names a user function to be called for each record examined during the search. The argument sfunc contains the address of the user function. The user function must return 1 if the record is accepted and Ø if the record is not accepted.

RETURNS

```
 Ø -- normal termination
-1 -- selection table is full
```

SEE ALSO

entsitm, mtchitm, clrfitm

UFSEL UFSEL

NAME

ufsel -- unisel function select

SYNTAX

```
ufsel (rec,count,tfunc)
long *count;
int (*tfunc) ();
```

DESCRIPTION

Use this function in the same manner as unisel. The difference is the user specifies the function to call for each selected record. Unisel uses a default function which updates a selection file for each record selected. Ufsel allows the user to specify an alternative, such as a print function.

ARGUMENTS

```
rec   -- the record type to select
count -- the address of a long to return the number of
         selected records
tfunc -- the address of a function to call for each
         selected record
```

RETURNS

```
 Ø -- normal return
-1 -- file contains no records
-2 -- one of the match items specified an invalid
      selection file
```

SEE ALSO

unisel

UNISEL UNISEL

NAME

unisel -- select database records

SYNTAX

```
unisel (selfile,rnum,count)
char *selfile;
long *count;
```

DESCRIPTION

Use this function to select records based upon selection criteria set up by calls to entsitm, mtchitm, or sfncitm. The result is a selection file, which contains all records that match the selection criteria. Calls to opensf, frstsel, nextsel, and closesf allow the user to retrieve the records. A normal sequence of events for using the function is as follows:

1. The user program describes the selection criteria by making calls to entsitm, mtchitm, and sfncitm.
2. A selection file name is generated and used in the call to unisel.
3. After a successful return, count is checked, and if greater than Ø, the selection file is opened using opensf.
4. Frstsel and nextsel are used to retrieve the selected records, and normal record level functions are used to update or print information in the records.
5. Closesf is used to close the selection file followed by a call to unlink.

ARGUMENTS

selfile -- the address of a null terminated string containing the name of the selection file for unisel to create

rnum -- the record type to select

count -- the address of a long which contains a count of records selected by unisel. Check the value after a successful return

RETURNS

Ø -- successful return
-1 -- the file to select upon is empty
-2 -- one of the match items has the wrong selection file name
-3 -- the selection file cannot be created

NOTES

Only one search can be set up at a time. See uqsrch and uqmtch.

Unisel uses the buffering routines, such as bseqacc, and so it is recommended that iniubuf be called in the user program to increase the size of the I/O buffer from the default size of 2K.

SEE ALSO

entsitm, mtchitm, sfncitm, clrsitm, clrfitm, uqsrch, uqmtch, opensf, frstsel, nextsel, prevsel, closesf, iniubuf

UQMTCH **UQMTCH**

NAME

uqmtch -- unisel quick match

SYNTAX

```
uqmtch (sf,field,rfield,msf,count)
char *sf, *msf;
long *count;
```

DESCRIPTION

Use this function to create a selection file based on one match item. Since the function does not use the unisel selection table, it can be called while the selection table is being built.

ARGUMENTS

sf -- a pointer to a string containing the name of the selection file to create

field -- the database field upon which the selection is based

rfield -- the referenced field in the record type of the match selection file

msf -- a pointer to a string containing the name of an existing selection file which will be used in matching records

count -- the address of a long to return the number of selected records

RETURNS

Ø -- normal return
-1 -- the match selection file does not exist or is not in the correct format
-4 -- the file contains no records
-6 -- the selection file cannot be created

SEE ALSO

unisel, mtchitm

UQSRCH UQSRCH

NAME

uqsrch -- unisel quick search

SYNTAX

```
#include "unisel.h"
uqsrch (sf,field,val1,val2,mode,count)
char *sf, *val1, *val2;
int  mode
long *count;
```

DESCRIPTION

Use this function to create a selection file based on one selection item. Since the function does not use the unisel selection table, it can be called while the unisel selection table is being built.

ARGUMENTS

sf -- a pointer to a string containing the name of the selection file to create

field -- the database field upon which the selection is based

val1 -- the address of a buffer containing the value of the field used in the selection

val2 -- the address of a buffer containing a second value, if the selection is a range. If the selection is not a range, set to Ø.

mode -- the mode of the selection: GT, GTE, LT, LTE EQ, NOT. The modes are defined in unisel.h.

count -- the address of a long to return the number of selected records

RETURNS

```
 Ø -- normal return
-1 -- invalid field
-2 -- vall is not set
-3 -- invalid mode
-4 -- the file contains no records
-6 -- the selection file cannot be created
```

SEE ALSO

unisel, entsitm

Explicit Relationship Functions

This section describes the host language interface to the explicit relationship functions. These functions allow you to follow the pointer lists maintained as a result of declaring an explicit relationship in the schema design. Retrieving related records, using pointers instead of B-trees, offers significant performance advantages when the records do not have to be retrieved in a predefined sequence.

Also provided is a set of higher-level functions that allow you to select and sort records based on the explicit relationship paths. These functions build a sorted tag file according to your specifications, which can then be traversed. This file gives you a way to retrieve records in a specific sequence and is particularly suited to report and batch update programs.

CLRSET CLRSET

NAME

clrset -- clear a previously identified set

SYNTAX

clrset (field, rfield)

DESCRIPTION

Use clrset to clear a set previously defined by a call to makeset. The field, rfield combination must match an existing entry in the table of active sets. The maximum number of active sets is currently 1Ø.

RETURNS

Ø -- the set has been cleared
-1 -- there is no such set active

FACCESS **FACCESS**

NAME

faccess -- access related record using specified field

SYNTAX

faccess (rnum,field)

DESCRIPTION

Use faccess to retrieve a related record using the value in a specified field. Rnum is the record to access, and field is the field containing the value. The record that contains the field must be current, or the program will be faulted. If the field is an explicit relational field to the specified record, the pointer is used. Otherwise, the record is accessed by key using the value in the field. Therefore, the field must be of the same type and length as the key of the specified record.

RETURNS

Ø -- access was successful, record of type rnum is current
-1 -- access unsuccessful, record is not current

MAKESET **MAKESET**

NAME

makeset -- identify a set of related records

SYNTAX

makeset (field, rfield, direction)

DESCRIPTION

Use this function to identify a set of related records for subsequent calls to nextrec, prevrec, and samerec. Makeset can only be used where explicit relationships have been defined in the schema. The parameters are as follows:

field:

The name of the key field of a record. A record of this type must be current when the function is called. The value of the key field determines which related records are retrieved. If you wish to retrieve all the records in a particular file, use the sequential access function, seqacc.

rfield:

The name of a field in another record that references the above field. Records whose data value in rfield match the data value in field are retrieved.

direction:

Indicates where to position the current record pointer for calls to nextrec and prevrec. There are two strings defined in file.h for use by programs for this purpose -- **first** and **last**. If using first, the pointer is positioned at the beginning, and the first call to nextrec returns the most recently added record to the file. If using last,

the pointer is positioned at the end of the file, and prevrec returns the oldest record in the file. If you wish to retrieve records in some other order, refer to the unisort function.

Using the example inventory schema, one would identify all models for the current manufacturer in LIFO (last in, first out) order with the following call:

```
makeset (mano, momano, first);
```

This call identifies the set of model records that have the same manufacturer number as the current manf record.

RETURNS

```
 Ø -- set has been established
-1 -- no such relationship
-2 -- no more space in the set table
```

NEXTREC NEXTREC

NAME

nextrec -- get the next record in a set

SYNTAX

nextrec (field, rfield)

DESCRIPTION

Nextrec makes the next record current from the set of records established by a previous call to makeset. The field, rfield pair must match a previous makeset call. Field is the key of some record, and rfield is an ordinary field in some other record that refers to it. The records are returned in LIFO (last in, first out) order as they were added to the file.

RETURNS

Ø -- the next record in the set is current
-1 -- no such set is currently defined
-2 -- end of the list has been reached

PREVREC **PREVREC**

NAME

prevrec -- get the previous record in a set

SYNTAX

prevrec (field, rfield)

DESCRIPTION

Prevrec makes the previous record current from the set of records established by a previous call to makeset. The field, rfield pair must match a previous makeset call. Field is the key of some record, and rfield is an ordinary field in some other record that refers to it. The records are returned in FIFO (first in, first out) order as they were added to the file.

RETURNS

Ø -- the previous record in the set is current
-1 -- no such set is currently defined
-2 -- beginning of the list has been reached

SAMEREC **SAMEREC**

NAME

samerec -- restore the current record of a specified set

SYNTAX

samerec (field, rfield)

DESCRIPTION

Samerec restores the current record of a specified set so the other database functions can operate on it. It is useful when manipulating several sets of records of the same record type. The field, rfield pair must match a previous makeset call. Field is the key of some record, and rfield is an ordinary field in some other record that refers to it.

RETURNS

Ø -- the record is now current
-1 -- no such set has been identified
-2 -- there is no current record for the specified set

SETSIZE SETSIZE

NAME

setsize -- determine the number of records in a set

SYNTAX

```
setsize (field, rfield, count)
long *count;
```

DESCRIPTION

This function returns the number of records in the specified set. The field, rfield pair must match a previous call to makeset.

RETURNS

```
 Ø -- the number of records is returned in count
-1 -- no such set is currently defined
```

SFRSTREC SFRSTREC

NAME

sfrstrec -- get the first record in a sorted set

SYNTAX

sfrstrec (field, rfield)

DESCRIPTION

Sfrstrec makes the first record current from the set of sorted records established by a previous call to unisort (or selsort, see below). The field, rfield pair must correspond to a previous unisort call. The field is the key of the record that passes to unisort, while rfield is the same as in the unisort call. If literal Ø is used as the record name in unisort, it must also be used as the field parameter to sfrstrec.

If the previously called function is selsort rather than unisort, refer to the selsort documentation for a description of the arguments to use.

RETURNS

Ø -- the first record in the set is current
-1 -- no such set is currently defined
-2 -- no records are in the set

SLASTREC SLASTREC

NAME

slastrec -- get the last record in a sorted set

SYNTAX

slastrec (field, rfield)

DESCRIPTION

Slastrec makes the last record current from the set of sorted records established by a previous call to unisort (or selsort, see below). The field, rfield pair must correspond to a previous unisort call. The field is the key of the record that passes to unisort, while rfield is the same as in the unisort call. If literal Ø is used as the record name in unisort, it must also be used as the field parameter to slastrec.

If the previously called function is selsort rather than unisort, refer to the selsort documentation for a description of the arguments to use.

RETURNS

Ø -- the last record in the set is current
-1 -- no such set is currently defined
-2 -- no records are in the set

SNEXTREC **SNEXTREC**

NAME

snextrec -- get the next record in a sorted set

SYNTAX

snextrec (field, rfield)

DESCRIPTION

Snextrec makes the next record current from the set of sorted records established by a previous call to unisort (or selsort, see below). The field, rfield pair must correspond to a previous unisort call. The field is the key of the record that passes to unisort, while rfield is the same as in the unisort call. If literal Ø is used as the record name in unisort, it must also be used as the field parameter to snextrec. The records are returned in the order specified by the sort key passed to unisort.

If the previously called function is selsort rather than unisort, refer to the selsort documentation for a description of the arguments to use.

RETURNS

Ø -- the last record in the set is current
-1 -- no such set is currently defined
-2 -- end of the list has been reached

SPREVREC SPREVREC

NAME

sprevrec -- get the previous record in a sorted set

SYNTAX

sprevrec (field, rfield)

DESCRIPTION

Sprevrec makes the previous record current from the set of sorted records established by a previous call to unisort (or selsort, see below). The field, rfield pair must correspond to a previous unisort call. The field is the key of the record that passes to unisort, while rfield is the same as in the unisort call. If literal Ø is used as the record name in unisort, it must also be used as the field parameter to sprevrec. The records are returned in the reverse order specified by the sort key passed to unisort.

If the previously called function is selsort rather than unisort, refer to the selsort documentation for a description of the arguments to use.

RETURNS

Ø -- the previous record in the set is current
-1 -- no such set is currently defined
-2 -- beginning of the list has been reached

UNISORT UNISORT

NAME

unisort -- select and sort records

SYNTAX

```
unisort(rnum,rfield,fldtbl,functbl)
int fldtbl[]
int (*functbl[])() {selfunc,confunc,srtfunc,tagfunc};
int selfunc(),confunc(),srtfunc(),tagfunc();
```

DESCRIPTION

Use this function to select and sort a set of records which will subsequently be visited by calls to snextrec or sprevrec. When the function returns, the sorting and selection are complete. Unisort allows a wide range of flexibility through the use of user defined functions.

Most of the work of unisort is done during the selection phase. It is at this point that it visits the specified set of records, accepts or rejects each record, and then writes a "tag" record to a temporary file. A tag record is an ASCII representation of the sort fields followed by the record location. After the tag file is created, a general sort function is invoked to order the records according to the keys in the tag records.

The arguments have the following uses:

rnum:

This record is the starting point of the sort. The desired record must be current before calling unisort. The first thing unisort does is call makeset, using the key of record rnum and rfield to identify the set of interest.

If the continuation function, confune(), is used, more than one occurrence of the rnum record type can be used in the sort. Unisort uses nextrec to visit each matching related record in the set. When the end of the list is reached, it calls the continuation function (if one is given) to see if there are any more sets from which to select and sort records.

If no explicit set of records is defined in the schema for the particular report requirement, all the records in the file must be passed. Unisort selects and sorts an entire file of records if you pass a literal Ø as the rnum parameter. This method of selection takes considerably longer than following an explicit relationship, especially if you need to select only a small number of records.

rfield:

Rfield is a reference field in some other record that refers to the key of the rnum record type. It identifies the target record in the sort and the path in the database to follow to access those records. If literal Ø is used as the rnum parameter, rfield can be the name of any field in the target record (the key field, for instance), instead of an explicit reference field.

fldtbl:

Fldtbl is an integer array of the fields which make up the sort key. It is in order of most significant to least significant. The fields in this list must be either in the target record or any direct or indirect parent of the target. (For example, they may be in the parent of a parent.) The list is composed of list elements separated by a comma and terminated with a -1. Each list element looks like the following:

field, rfield, rfield, ... Ø,

The field is the sort key, while the following list of Ø terminated rfields describes the path to the record that contains the field. If the field is in the target record,

the rfield list is empty.

functbl:

This argument is an array of functions called by unisort various times during the selection phase. These functions let the user specify the subset of records and override the default tag creation methods of unisort. The functions and their uses are described below:

selfunc -- This function lets the user's program specify which target records to include in the sort. Each time a new target record is visited, unisort calls the selection function. The function can perform whatever processing is necessary to determine whether to accept or reject the record. It must return 1 to select the record, and Ø to reject it. If the selection function address in the functbl array is Ø, the function is not called, and unisort selects all the records.

confunc -- This function lets the user change the current rnum record and then continue the selection. The user can then combine several sets of target records for one sort. The following schema illustrates the point:

SCHEMA REPORTS
Schema Listing

RECORD/FIELD	REF	TYPE	LEN	LONG NAME
month	12			
*mno		NUMERIC	4	
mname		STRING	1Ø	
day	365			
*dday		DATE		
dmon	mno	NUMERIC	4	
appt	4ØØØ			
*apno		NUMERIC	5	
apname		STRING	3Ø	
aptime		TIME		
apday	dday	DATE		

This schema design is an appointment scheduling database. There can be several appointments for any given day, and you can access all of the appointments for a day by using makeset (dday, apday, 1) and nextrec. Suppose, you want to sort all the appointments for the month of January by appointment name (apname). You can use unisort (node, appt,.....), but this requires accessing all of the appt records in the file. It is much better to select only the day records for the month of January and then combine all of the appt records belonging to the day records for one sort by apname. To do this, use the continuation function. The main program accesses the January month record (mno=1). The program then does a makeset to identify all the day records in January, a nextrec to access the first day record, and calls unisort with a continuation function only. (A selection function is needed only if you are selecting on some other criterion than date, such as appointments before 1Ø a.m.) The continuation function uses nextrec to obtain the next day and returns 1 as long as there are more day records. It returns Ø when the end of the list is reached, and the selection phase then ends.

It is also possible to use the continuation function to

do additional selection, such as appointments scheduled on Thursdays. (This is accomplished by having return 1 only on desirable records and skipping the rest.) In general, the continuation function allows a greater reduction in the number of records visited in order to complete the selection phase.

srtfunc -- Use this function to change the selected sort fields during the selection process. You can use it when the records' sort value is dependent on various fields. For example:

SCHEMA REPORTS
Schema Listing

RECORD/FIELD	REF	TYPE	LEN	LONG NAME
expen	3ØØØ			expenditure_item
*exno		NUMERIC	6	id_number
edesc		STRING	3Ø	description
ebamt		AMOUNT	7	budgeted_amount
eaamt		AMOUNT	7	actual_amount
edate		DATE		expense_date

Each expen record represents a budgeted expenditure for some corporation. Suppose, you want a list of all expenditures either planned or made between two given dates. (In the case of a planned expenditure, you must use the budgeted amount, but in the case of a completed expenditure, use the actual amount for the sort.) To accomplish this, include a sort function which modifies the field table (fldtbl), depending on the current record's status. If the actual amount field is null (see intro), it places ebamt in the first word of the field table. Otherwise, it places eaamt there. Two things are important to note. First, the field table must be initialized with fields of the same type and length), even when the srtfunc is used. Second, a field may only be replaced with another field of the same type by srtfunc. A Ø in place of srtfunc results in the initialized field list being used throughout.

tagfunc -- The tag function lets the user create his own tag

for any given record. This function is useful when the records are sorted by a computed field. Unisort passes the function a pointer. The function places its desired tag (in ASCII form) into the buffer and returns the number of characters. It must always return tags of the same length. When there is a tag function, the field table is ignored, and only the returned tag is used.

RETURNS

Ø -- records are sorted
-1 -- selection file is empty

Secondary Index Functions

This section describes the B-tree secondary index functions available from a host language. A B-tree index is created in either ascending or descending order for any database field, in order to improve performance when doing queries on that field. An index can also be established if records frequently need to be retrieved in sequence by the specified field, and you don't want to use an explicit relationship and unisort as described in "Explicit Relationship Functions" in this chapter. In general, establishing a B-tree index improves the retrieval performance of a database, while slowing down updating of the indexed field.

A B-tree index search consists of two basic operations: search, and get next. The search operation makes an initial record current. It also indicates whether or not an entry was found that matches the search argument exactly. The get next operation is used to step through the index, beginning just after the position found by a successful search. If duplicate field occurrences exist, a search, followed by one or more get next operations, accesses all of the records which have the field value in common.

The fields that make up a combination field can be indexed, but the resulting combination field cannot. Unify automatically maintains all B-tree indexes to ensure data integrity when many users are simultaneously updating and retrieving records. Indexes built for string fields of length greater than 80 do not consider characters beyond the 80th position.

BTNEXT **BTNEXT**

NAME

btnext -- get next record using a B-tree index

SYNTAX

btnext(field)

DESCRIPTION

Btnext gets the next field occurrence in the specified field's B-tree index and makes the record current. The index must have been opened for searching by function opnbts. You must use function btsrch to search into the index (get first) before calling btnext.

RETURNS

-3 -- the specified field's index is not open for searching
-2 -- function btsrch was not called prior to this function
-1 -- end of index encountered -- a record was not made current
Ø -- the record is current

Since this function uses look-ahead buffering, records that are being added to the database while it is being used may be missed.

BTSRCH BTSRCH

NAME

btsrch -- search a field's B-tree index

SYNTAX

```
btsrch(field,buf)
char *buf;
```

DESCRIPTION

Btsrch searches a field's B-tree index that you opened for searching by function opnbts and makes a record current. Buf is the address of an occurrence of field in internal form. This field occurrence is the search argument. The search target is the field occurrence in the index that is logically just beyond the search argument or equal to it. After btsrch has searched the index, use function btnext to traverse it.

RETURNS

-2 -- the field's B-tree index has not been opened for searching by function opnbts
-1 -- the search argument is logically beyond all field occurrences in the index -- a record was not made current
Ø -- the occurrence found does not match the search argument -- the record is current
1 -- the field occurrence found matches the search argument -- the record is current

CLOSBT CLOSBT

NAME

closbt -- close a B-tree index

SYNTAX

closbt(field)

DESCRIPTION

Closbt closes the specified field's B-tree index, which was opened for searching by function opnbts.

RETURNS

-1 -- the specified field's index (if one exists) was not open for searching
Ø -- the specified field's index is closed

FLDIDXD FLDIDXD

NAME

fldidxd -- determine if a field is indexed

SYNTAX

fldidxd(field)

DESCRIPTION

Fldidxd determines whether or not a field is indexed in a B-tree index, and if it is indexed, whether the index is in ascending or descending order.

RETURNS

Ø -- field is not indexed in a B-tree
1 -- field is indexed in ascending order
2 -- field is indexed in descending order

OPNBTS OPNBTS

NAME

opnbts -- open a B-tree index for searching

SYNTAX

opnbts(field)

DESCRIPTION

Opnbts opens a field's B-tree index for searching. An index must exist for the particular field. Once an index is open, it can be searched repeatedly. Functions btsrch and btnext are used to search into and then (optionally) step through a field's B-tree index, which has been opened for searching. Up to five indexes can be open concurrently for searching.

RETURNS

-3 -- the specified field is not indexed in a B-tree index
-2 -- maximum number of indexes are open
-1 -- the specified field's index is already open
Ø -- the specified field's index is open for searching

Buffered Sequential Access Functions

This section describes the host language interface to the buffered sequential access functions. These functions let you scan a file using raw I/O and large buffers when there is no other method (hash, B-tree, or explicit relationship) of accessing records. This method permits very rapid processing of all the records in a file and is preferred, even if there is another access method, when you need to examine every record. This method is preferred because the disk head movement is at a minimum when reading in very large blocks, and seek time is by far the bottleneck in most cases when selecting records.

Since you are using raw I/O data, these functions are only suitable for read-only usage, such as reporting or selecting records. The data is read into the private data space of the program, and if you make any updates in this buffer, there can be conflicts with the data in the TRS-XENIX kernel buffers. For this reason, no update functions are provided in this package. Once a record is made current using bsetloc, bseqacc or bfaccess, you can use the standard record level update functions.

Many of the functions described in this section produce similar error messages as their unbuffered versions in the Record Access Functions section.

BGFIELD BGFIELD

NAME

bgfield -- buffered version of gfield

SYNTAX

```
bgfield (field,buf)
char *buf;
```

DESCRIPTION

This function retrieves a field from the database and stores it in a buffer. This version uses a buffering scheme and can only be called after the record is made current using **bseqacc.**

RETURNS

none

SEE ALSO

gfield, bseqacc, bfaccess, iniubuf

BSEQACC **BSEQACC**

NAME

bseqacc -- buffered version of seqacc

SYNTAX

bseqacc (rnum, direction)

DESCRIPTION

Use this function to retrieve all records in a file in the order in which they are stored. This version uses a buffering scheme to increase the speed of the retrieval. Rnum is the record type to retrieve, while "direction" is chosen from the following set:

first -- access the first record in the file
next -- access the next sequential record in the file
last -- access the last record in the file
prev -- access the previous sequential record in the file

RETURNS

Ø -- the specified record is current
-1 -- the specified record does not exist

SEE ALSO

seqacc, bgfield, bfaccess, iniubuf

BSETLOC BSETLOC

NAME

bsetloc -- buffered version of setloc

SYNTAX

```
bsetloc (rnum,loc)
long loc;
```

DESCRIPTION

This function lets the user set the current record for a given record type. This version is used with the buffering routines. If a record is not accessed using bseqacc, then you must call this function before making calls to bgfield and bfaccess. Rnum is the record type and loc is the record location which was acquired earlier from a call to loc.

RETURNS

none

BFACCESS BFACCESS

NAME

bfaccess -- buffered version of faccess

SYNTAX

bfaccess (rnum,field)

DESCRIPTION

Use this function to retrieve a related record using the value in a specified field. Rnum is the record to access and field is the field from which the value comes. This version uses a buffering scheme, and so the record which contains "field" must be made current using bseqacc. The remaining description of faccess applies here.

RETURNS

Ø -- the access was successful, record of type rnum is current
-1 -- the access was unsuccessful

SEE ALSO

faccess, bseqacc, bgfield, iniubuf

INIUBUF INIUBUF

NAME

iniubuf -- initialize the read buffer for bseqacc and other buffering routines

SYNTAX

```
iniubuf (bufaddr, bufsz)
char *bufaddr;
int bufsz;
```

DESCRIPTION

This function lets the user specify the read buffer for the buffering routines: bseqacc, bgfield, and bfaccess. If this routine is not called, a default buffer with a size of 2K bytes is allocated using **sbrk**. Call iniubuf at the beginning of your program, and make sure the size of the buffer is as large as possible to increase the speed of sequential searches of the database.

ARGUMENTS

bufaddr -- the address of a buffer allocated in the user program
bufsz -- the size in bytes of the buffer

RETURNS

none

SEE ALSO

bseqacc, bgfield, bfaccess

Terminal I/O Functions

This section describes the host language interface to the terminal I/O functions. These functions display data and prompts on the screen and get input from the user. There are two levels of functions provided -- those that interface to screen forms defined using SFORM, and a lower level of functions that use x-y coordinates and literal text to interact with terminals. All of these functions use the termcap library functions to perform terminal independent I/O.

For the SFORM functions described, ssfld and esfld are screen field names as defined in the .h file produced by the Process Screen program (Chapter 4). Ssfld is the starting screen field number, which must come before the ending screen field number, esfld.

CLEANCRT **CLEANCRT**

NAME

cleancrt -- clear the foreground

SYNTAX

```
cleancrt()
```

DESCRIPTION

This function clears the screen foreground. There are two methods, one for terminals with this hardware feature and one for terminals without this feature. On terminals with clear foreground, this function simply clears the foreground, and only the screen field prompts remain. On other terminals, this function makes calls to erasprmp for every screen field. You must erase the "garbage" on the screen separately.

RETURNS

none

Unexpected results may occur if loadscr is not used to load the screen form.

CLEARSCR **CLEARSCR**

NAME

clearscr -- clear the screen

SYNTAX

clearscr()

DESCRIPTION

This function clears the screen starting at line 3. The net effect is to leave the heading that appears in the menu handler and blank the rest of the screen. When used in conjunction with **loadscr**, multiple page screen forms can be implemented.

RETURNS

none

DSPLY DSPLY

NAME

dsply -- display data for screen fields

SYNTAX

dsply(ssfld,esfld)

DESCRIPTION

This function displays the data associated with a set of screen fields. Starting with ssfld and ending with esfld, the function goes down the screen calling output for each one. Note that this function always displays data from the database, so the required records must be current to avoid generating a fault.

RETURNS

none

ERASPRMP **ERASPRMP**

NAME

erasprmp -- erase data for screen fields

SYNTAX

erasprmp(ssfld,esfld)

DESCRIPTION

This function erases the data portion of a set of screen fields from the screen. Starting with ssfld and ending with esfld, it goes down the screen writing blanks over the maximum length of each screen field.

RETURNS

none

GDATA **GDATA**

NAME

gdata -- get field data from screen

SYNTAX

```
gdata(fx,fy,field)
int fx,fy;
```

DESCRIPTION

Use this function to take data from a specified location on the screen, validate it, convert it, and then store it directly in the database. The record into which the new field is being inserted must be current. Only UNITRIEVE related checks are made on the field. For example, if a field is NUMERIC 3 in the schema, it is validated for three numeric characters before being stored. If additional checks are needed, use **gtube** to take in the field before storing it.

The fx and fy arguments are integers containing the location of the field on the screen. Fx is the column, which ranges from Ø to 79. Fy is the row, which ranges from Ø to 23.

This function uses pfield to store the field. Therefore, the data is stored in the current record of the type containing the field.

If the user enters the new action character (usually an <F2>), gdata immediately returns -2, and no data is stored in the field. If the user presses <ENTER>, no data is stored in the field, and the function makes a normal return.

When entries are made which do not correspond to the field type (alpha in a numeric field), an error message is printed, and the cursor is repositioned at the start of the field.

RETURNS

Ø -- the entry has been accepted
-2 -- a new action return was taken

GTUBE GTUBE

NAME

gtube -- input a field from the screen into a buffer

SYNTAX

```
gtube(fx,fy,field,buf)
int fx,fy;
char *buf;
```

DESCRIPTION

This function works much like gdata except the data is stored in a buffer instead of the database. The field is taken in from the screen at the specified location, checked for basic UNITRIEVE validity, converted to internal format, and then stored in the indicated buffer. The buffer should be of a type to accept the incoming field as follows:

NUMERIC	(1-4)	short
	(5-9)	long
FLOAT		double
AMOUNT	(1-7)	long
	(8-12)	double
DATE		short
TIME		short
STRING		char buf[?]
COMB		a structure that mirrors the structure of the field components

This function is particularly useful when more than the standard UNITRIEVE edit is to be performed. In this case, you can call gtube to input the field, perform additional validity checks, and store the field using pfield, if the field is valid.

If the user inputs the new action character, <F2>, a -2 is returned immediately from the function. If <ENTER> is pressed, a -3 is returned, and the contents of the buffer are undefined.

RETURNS

Ø -- the field just entered is in the buffer
-2 -- a new action return was taken
-3 -- <ENTER> was pressed

INBUF INBUF

NAME

inbuf -- input a screen field into a buffer

SYNTAX

```
inbuf(sfld,buf)
char *buf;
```

DESCRIPTION

This function lets the user input a screen field into a buffer. The data is in internal format after the call. The screen coordinates and the field type are gathered from the screen field name (sfld). Editing related to the valid external format of the data is performed, since this function uses gtube. The data is stored in buf.

RETURNS

```
-3 -- <ENTER> was pressed
-2 -- <F2> was entered
 Ø -- a valid field was input and stored in buf
```

INPUT **INPUT**

NAME

input -- input a field

SYNTAX

```
input(sfld)
```

DESCRIPTION

This function inputs a field directly from a screen field and stores it in the database. The x and y coordinates and the field type are all gathered from the screen field name (sfld). The record type containing the field specified by sfld must be current. Since this function uses gdata, editing related to the valid external format of the data is performed.

RETURNS

```
 Ø -- the field has been input and accepted
-2 -- <F2> was entered
```

LOADSCR LOADSCR

NAME

loadscr -- display screen, loading parameters into memory

SYNTAX

```
loadscr(scrn)
char *scrn;
```

DESCRIPTION

Use this function to display a screen form and load its parameters into memory. It must be called before calling other functions that use screen fields. Scrn is the null terminated name of the screen to be used. If the screen doesn't exist or if any of the database fields specified by the screen fields don't exist in the schema, loadscr aborts.

RETURNS

none

OUTBUF OUTBUF

NAME

outbuf -- display buffer to a screen field

SYNTAX

```
outbuf (sfld,buf)
char *buf;
```

DESCRIPTION

This function takes the data from a buffer, formats it for output, and displays it at a given screen field. The screen location and field type are all gathered from the screen field name (sfld). The data is taken from buf.

RETURNS

none

OUTPUT OUTPUT

NAME

output -- display a field

SYNTAX

output (sfld)

DESCRIPTION

This function outputs a field directly from the database to the data position of the specified screen field (sfld). The x and y coordinates and the field type are determined from the screen field name. The record type containing the database field specified by sfld must be current.

RETURNS

none

PDATA **PDATA**

NAME

pdata -- print data from the database to the screen

SYNTAX

```
pdata(fx,fy,field)
int fx,fy;
```

DESCRIPTION

This function takes a field from the database, converts it for output, and then displays it at a specified location on the screen. Fx and fy are the x and y coordinates, respectively, where the data is displayed on the screen. Since gfield is used to obtain the data, the field is taken from the current record of the type containing the field.

RETURNS

none

PRMP PRMP

NAME

prmp -- output a string to the screen

SYNTAX

```
prmp(x,y,str)
int x,y;
char *str;
```

DESCRIPTION

Use this function to display a string at a specific location on the screen. The first two arguments are the x and y coordinates, indicating the starting point of the string. The last argument is the null terminated string to be displayed.

RETURNS

none

PRTMSG PRTMSG

NAME

prtmsg -- display a message and wait for operator verification

SYNTAX

```
prtmsg(x,y,strng)
int x,y;
char *strng;
```

DESCRIPTION

This function displays strng at x,y on the screen and waits for the operator to enter any keyboard response. The string is erased after the response is entered. It is useful for displaying error messages.

RETURNS

none

PTUBE PTUBE

NAME

ptube -- write a field to the screen from a buffer

SYNTAX

```
ptube(fx,fy,field,buf)
int fx,fy;
char *buf;
```

DESCRIPTION

This function is very similar to pdata, except the data is taken from a buffer, not from the database. The function converts the data in the buffer to the external format and then displays it on the screen. See gtube for the type of buffer needed to hold each UNITRIEVE field type.

This function is useful if the actual data to be displayed is a computed total. Simply pick a field with the same type as the computed total, and use it as the field argument.

RETURNS

none

SFLDESC SFLDESC

NAME

sfldesc -- screen field description

SYNTAX

```
#include "fdesc.h"
sfldesc (sfield,sfdsc)
int sfield;
SFLDESC *sfdsc;

The following is a description of SFLDESC:

#define SFLDESC struct sfdesc
struct sfdesc {
     int sf_fld;    /* associated database field */
     int sf_col;    /* column number */
     int sf_lin;    /* line number */
};
```

DESCRIPTION

This function returns the attributes of a screen field in sfdsc. Before using this function, you must use the loadscr function.

ARGUMENTS

```
sfield -- the screen field
sfdsc  -- the address of a SFLDESC structure
```

RETURNS

```
Ø -- invalid screen field
1 -- normal return
```

SEE ALSO

fldesc

YORN YORN

NAME

yorn -- prompt for a yes/no response from operator

SYNTAX

```
yorn (str)
char *str;
```

DESCRIPTION

This function displays the indicated string on line 22 (column 1) of the screen and waits for either a Y or N response from the operator (lowercase response is also accepted). If an unknown response is given, another message prompting for a y or n appears on line 23.

RETURNS

```
 Ø -- "N" or "n" was entered
 1 -- "Y" or "y" was entered
-2 -- something other than "y" or "n" was entered
```

Printer I/O Functions

This section describes the host language functions that send database information to the printer. These functions make it easy to format reports exactly the way you want them.

FLUSH **FLUSH**

NAME

flush -- output print buffer

SYNTAX

```
flush(fd)
int fd;
```

DESCRIPTION

This function outputs the print buffer to the user specified file descriptor. The print buffer should be initialized (see oblank) and then filled using odata, obuf, prstr, etc. The file descriptor is the only argument, and it should be returned by an earlier call to open or oprf.

After the call to the function, the print buffer is cleared, so that further calls to flush produce blank lines. It is also interesting to note that only the number of characters formatted on a line are actually printed; there are no trailing blanks.

RETURNS

none

OBLANK OBLANK

NAME

oblank -- initialize print buffer

SYNTAX

oblank()

DESCRIPTION

Call this function at the start of a program, which does printer output, to initialize the print buffer. You only need to call this function one time in any executable, but it must be called prior to the first database output call.

RETURNS

none

OBUF OBUF

NAME

obuf -- output from a buffer to the printer

SYNTAX

```
obuf(col,field,buf)
char *buf;
in col;
```

DESCRIPTION

This function formats the contents of a buffer and places the results into the current print line for later output by flush. The data is converted according to the type of the "field" argument and stored in the print buffer at column col.

RETURNS

none

ODATA ODATA

NAME

odata -- output a database field to the printer

SYNTAX

```
odata(col,field)
int col;
```

DESCRIPTION

This function formats a field into the current print line for later output by **flush**. The field is taken from the database in the same manner as pdata and then placed into the print line in column col.

RETURNS

none

PFORM **PFORM**

NAME

pform -- print special forms

SYNTAX

```
pform (xbuf,ofd,pftab)
  char xbuf[];
  int ofd;
  struct pftable
          {
           int (*func) ()
           char *str;
          } pftab[];
```

DESCRIPTION

Pform is designed to allow the format of reports to be specified in an external text file that is read at run time. Database fields, literal strings and user defined function results can be placed at x-y coordinates specified in the external file. This function lets format changes be made without recompiling or reloading the program; therefore, a user can make these changes without the assistance of a programmer.

To use pform, call the function pform, which then prints a single copy of the form that is described in the text file. Xbuf is a string that contains the name of the external format file. Ofd is the output file descriptor returned from an open or create call. Pftab is a table of user defined function addresses and names for printing special strings.

The use of pform is best illustrated by an example. On the following pages is a simple program to print inventory labels using pform, based on the inventory schema introduced in the Tutorial Manual.

```
#
/*****************************************************************
*         inv3ØØ.c
*
*         This is a simple report program for the inventory
*         system that prints labels to stick on parts in the
*         warehouse. It uses pform to position the output.
*
*
*         arguments:
*
*         returns:
*
*    author: wao
*    date: Ø6/23/82  14:12
*****************************************************************
$include "../../def/file.h"

extern int pdate (); /* user function to print today's date */

struct {
        int (*func) ();
        char *str;
       } pftab []={
                    pdate, "pdate   ",
                    Ø,Ø
                  };

main ()
{
     int fd;        /* printer file descriptor */

     fd = oprf ();       /* open print spool file */
     if (seqacc (item, first) == Ø)
     do
     {
          faccess (model,imodel);
          faccess (manf, momano);
          pform ("inv3ØØ", fd, pftab);
     } while (seqacc (item, next) == Ø);
}
```

```
#
/*************************************************************
*
*         pdate.c
*
*         User function for pform to print today's date.
*
*         arguments:
*
*         returns:
*
*    author:wao
*    date: Ø6/23/82  15:41
*************************************************************
#include <time.h>

pdate (xbuf, xarg)
char xbuf[], xarg [];
{
     long tvec;          /* for time in seconds since 197Ø */

     struct tm *localtime(),
               *t;       /* for result of localtime call */

     time (&tvec);       /* get today's date */
     t = localtime (&tvec);
                    /* convert it to something more meaningful */

     sprintf (xbuf, "%2d/%2d/%2d", (t->tm_mon)+1, t->tm_mday,
          t->tm_year);   /* place the date in the pform output
                            buffer */
```

Text of the report format file (inv3ØØ.p):

```
6
1,1,SERIAL #
9,1,sno
1,2,MODEL #
8,2,imodel_monum
1,3,MANUFACTURED BY
17,3,mname
1,5,DATE LABELED:
14,5,pdate
```

Sample label:

```
SERIAL # 991234567
MODEL #1122333
MANUFACTURED BY INTERNATIONAL WIDGET CO.

DATE LABELED: 6/23/82
```

This program prints a label for each item in inventory. Pform expects to find the report format file in the directory, where the program is executed, in a file named inv3ØØ.p.

Pform considers the top left corner of the form to be the x-y coordinates 1,1. The x coordinate increases to the right, while the y coordinate increases as the form is advanced. The first line in the file has the length of the form, in this case, six lines. This length must include lines that are not printed, so pform will know how many line feeds to perform to reach the top of the next form. Each subsequent line defines a position of the form where text is to be printed. These lines have the following form:

x, y, <string>

x is the x coordinate of the text, and y is the y coordinate of the text; the string is either:

1. literal text
2. UNITRIEVE database field name
3. <function name>,<parameter string>

Line 2 is an example of a literal text string. It is imperative that you do not include commas (,) in your text string, since commas are recognized as delimiters in pform. The string "SERIAL #" is printed at column 1 on line 1. Line 3 is an example of a UNITRIEVE database field name. The record that contains the field must be current when pform is called; otherwise, a UNITRIEVE fault is generated. The contents of the field sno are printed at column 9 on line 1. If the contents of the field are shorter than the length defined in the schema, you must allow for the full schema-defined field length.

Line 9 is an example of a function name, in this case, with no parameter. The function name is the same as defined in pftab in the program. Pform searches pftab for the name of the function, which must be eight characters or less. The parameter can be a string of characters but cannot contain a blank. The user function is responsible for parsing the parameter. Pform calls the user function with two arguments as follows:

```
(*pftab[i].func) (pfstr, pfarg)
 char pfstr[];
 char pfarg[];
```

Pfstr is a global array, in which the user function copies the string to be printed, and pfarg is the parameter string. There are no restrictions on what the user function can do, but the maximum length of the output string is 128 characters.

Note that the lines in the format file are sorted by x coordinate within y coordinate. This makes sense if you think of how a report is printed on a teletype -- from left to right, with no backtracking.

PRSTR PRSTR

NAME

prstr -- output a string to the print buffer

SYNTAX

```
prstr(str,col)
char *str;
int col;
```

DESCRIPTION

This function lets the user output a fixed string to the print buffer for later output by **flush**. The first argument is a null terminated string. The second argument is the column for output.

RETURNS

none

Utility Functions

This section describes a set of general purpose application functions that can be used in writing programs. They include functions to perform character manipulation, terminal control, and resource locking.

CFILL **CFILL**

NAME

cfill -- fill a string with a character

SYNTAX

```
cfill(c,str,len)
char c,*str;
int len;
```

DESCRIPTION

This function fills an array (str) with a specified character. The first argument is the character to be placed in the array. The second argument is a pointer to the array which is to be filled. The third argument is the number of slots to fill.

RETURN

none

CLR_CRT CLR_CRT

NAME

clr_crt -- clear the terminal

SYNTAX

```
clr_crt(fd,flg)
int fd;
char flg;
```

DESCRIPTION

Use this function to clear the terminal. The first argument is the output file descriptor. The second argument is an 'a' if both background and foreground are to be erased. Otherwise, the second argument should be blank.

RETURNS

none

ERAS_LN **ERAS_LN**

NAME

eras_ln -- erase a line on the terminal

SYNTAX

```
eras_ln(fd)
int fd;
```

DESCRIPTION

This function issues the erase line command to the terminal. The argument is the file descriptor of the terminal.

RETURNS

none

GLOB GLOB

NAME

glob -- compare a string with a meta character mask

SYNTAX

```
glob(strng,mask)
char *strng,*mask;
```

DESCRIPTION

This function compares a string with a special string mask. The string mask has the same form as the command word argument described in "File Name Generation" in the description of the shell in section 1 of the TRS-XENIX System Reference Manual (SH (1)).

The mask is searched for the characters *, ?, and [. If one of these characters appears, the mask is regarded as a pattern. Following is what each of these special characters represents:

*	match any string including the null string
?	match any single character
[..]	match any one of the characters enclosed. A pair of characters separated by - matches any character lexically between the pair.

RETURNS

Ø -- the string does not match the mask
1 -- the string matches the mask

IVCMP IVCMP

NAME

ivcmp -- compare two arrays

SYNTAX

```
ivcmp(ptr1,ptr2,length)
char *ptr1,*ptr2;
int length;
```

DESCRIPTION

This function compares two character arrays to determine if they are the same. Ptr1 and ptr2 are pointers to these arrays. Length is the number of characters to compare.

RETURNS

Ø -- the strings do not compare
1 -- the strings are the same

KDATE KDATE

NAME

kdate -- convert from julian date

SYNTAX

```
kdate(jday,arr)
int jday,arr[3];
```

DESCRIPTION

This function converts a date from a julian sequential integer format into an integer array containing the month, day, and year. The first argument is the julian date, and the second argument contains the converted date after returning.

RETURNS

none

KDAY KDAY

NAME

kday -- convert to julian date format

SYNTAX

```
kday(arr)
int arr[3];
```

DESCRIPTION

This function is used to convert an integer array containing the month, day, and year into a julian sequential integer. The only way of knowing whether the date is valid is to convert the resulting julian date back to an integer array and compare with the original. If they compare, the date is valid.

RETURNS

the julian date

KEYBRD **KEYBRD**

NAME

keybrd -- lock or unlock the terminal keyboard

SYNTAX

```
keybrd(fd,flg)
int fd,flg;
```

DESCRIPTION

Use this function to lock or unlock the terminal keyboard. The first argument is the output file descriptor. The second argument specifies whether the keyboard is locked or unlocked. If it is not zero, the keyboard is locked.

RETURNS

none

LASTCHR LASTCHR

NAME

lastchr -- locate the last character in a string

SYNTAX

```
lastchr(strng,len)
char *strng;
int len;
```

DESCRIPTION

This function returns the index (Ø based) of the first blank or null character at the end of a string. It starts at the end of the string and scans until it finds a non-blank or non-null character. Its arguments are a pointer to the string and a maximum length.

RETURNS

The index of the first blank or null character following the end of the string

LEN **LEN**

NAME

len -- determine a string's length

SYNTAX

```
len(str)
char *str;
```

DESCRIPTION

This function returns the length of a null terminated string. The argument is a pointer to the string.

RETURNS

The length of the string

LOCK LOCK

NAME

lock -- lock a resource

SYNTAX

```
lock(c)
char c;
```

DESCRIPTION

Use this function to lock a resource from all the other users on the system. Use it when you don't want to lock out all database updates, which is the effect of using **startrans**. The argument to lock is simply an ASCII character representing the resource to be locked. The user establishes his own set of characters to represent various resources. UNIFY already uses some letters, as noted below.

If the same user locks the same resource twice, he must unlock a resource the same number of times, in order to free it for other users. Lock can be useful in situations where two users are sharing such things as a special purpose dedicated printer.

If another user attempts to get the same lock, lock sleeps for one second and then tries again. After five seconds, the message "Waiting for lock" is displayed at line 23 on the screen. If lock is unable to proceed after waiting 3Ø seconds, it exits, and control is returned to the parent process, usually the menu handler. The assumption in this case is that a lock file has been left sitting around by a process that died and needs to be removed.

For systems without resource locking implemented in the kernel, UNIFY uses this function to control concurrent database access. The lock function works by creating files in the current directory (or $DBPATH, if this environment variable is set) named

LOCKx, where x is the character passed to the function. UNIFY uses both d, for the database file, and u, for the data dictionary file (unify.db). In addition, the reconfiguration program uses k when backing up the data dictionary. The following table summarizes the locks that UNIFY uses.

File	Meaning
LOCKd	UNIFY database file (file.db) is being updated at the time of the failure and must be restored. The file must be removed before processing can proceed. This approach is only used when kernel resource locking is not available.
LOCKu	Data dictionary file (unify.db) is being updated at the time of the failure and must be restored. The file must be removed before processing can proceed. This approach is only used when kernel resource locking is not available.
LOCKk	Internal lock used by reconfiguration and create database. The program recovers unify.db from unify.bu when it runs again. Do not remove this file. UNIFY always uses this lock, even with kernel locking.
LOCKl	Lock used by the program loading function of the menu handler (lfilegen). This is used to prevent simultaneous use of the file a.out that the loader uses. If lfilegen is interrupted, LOCKl in the build directory must be removed before loading can be performed. UNIFY always uses this lock, even with kernel locking.

This resource locking approach is similar to the TRS-XENIX print spooler, which creates a file called /usr/spool/lpd/lock. There are two very important things to note about this approach:

1. Any and all UNIFY users MUST be ordinary TRS-XENIX

users -- not the super user! The super user is able to create the lock file even if one exists, thus defeating the concurrency control.

2. All users that access the database file MUST be located in the same directory as the database file (i.e. the working directory is the one which contains the database file), or they must all have $DBPATH set to the same directory. Users with different working directories are not locked against each other.

If there is a crash or if a program aborts or is faulted, one of these files may exist and have to be removed before proceeding. The presence of a LOCKx file after failure indicates that the associated database must be recovered from a backup, as an update operation was in progress at the time of the failure.

RETURNS

none

MV_CUR MV_CUR

NAME

mv_cur -- set the cursor position

SYNTAX

```
mv_cur(fd,x,y)
int fd,x,y;
```

DESCRIPTION

Use this function to move the terminal cursor to a specified location. The first argument is the output file descriptor. The second and third arguments are the x and y coordinates, indicating the location of the cursor.

RETURNS

none

OPRF OPRF

NAME

oprf -- create a pipe to lpr

SYNTAX

```
oprf()
```

DESCRIPTION

This function is used by programs wishing to output through the print spooler lpr. The function returns a file descriptor to write to the pipe.

RETURNS

The output file descriptor
-1 -- the pipe could not be created

PRIAMD PRIAMD

NAME

priamd -- prompt for operational mode

SYNTAX

```
priamd (line)
int line;
```

DESCRIPTION

This function interfaces with the menu handler security system to get the desired mode of operation for interactive database maintenance programs. It uses parameters passed to the sysrecev function by the menu handler to determine the current user id and update privilege. Programs that call priamd must be loaded with sysrecev as the main entry point in the executable. (See Chapter 2 for more details.) Priamd displays a prompt at line 22 on the screen, with options chosen from the following possibilities:

[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE

The options actually displayed are determined by the current user's update privilege for the program being executed. Any combination of the above privileges may be specified for each user.

When the user makes a valid selection from those displayed, the mode selected is displayed on the screen at x-y coordinates (1, line). The user can then see the mode of the current program. Control is then returned to the calling function. The calling function uses the return status value to perform the appropriate maintenance operation. If you receive the status code for <F2> (-2), the calling function returns control to the menu handler by exiting.

The global variable int logacl is set by sysrecev to the current user's access level. The format of the word is as follows:

```
logacl & Ø1Ø -- inquire privilege
logacl & ØØ4 -- add privilege
logacl & ØØ2 -- modify privilege
logacl & ØØ1 -- delete privilege
```

RETURNS

```
-2 -- <F2> was entered
 Ø -- inquire mode
 1 -- add mode
 2 -- modify mode
 3 -- delete mode
```

PTCT_CRT PTCT_CRT

NAME

ptct_crt -- modify terminal protect mode status

SYNTAX

```
ptct_crt(fd,flg)
int fd,flg;
```

DESCRIPTION

Use this function to take the terminal in and out of protect mode. If the terminal is in protect mode, you cannot write over write-protected characters. In addition, erases do not erase these characters. The first argument is the terminal file descriptor. The second argument is a flag indicating whether protect mode is being turned on or off. Any nonzero value turns protect mode on.

RETURNS

none

PTCT_WRT PTCT_WRT

NAME

ptct_wrt -- modify terminal write protect status

SYNTAX

```
ptct_wrt(fd,flg)
int fd,flg;
```

DESCRIPTION

Use this function to modify the write protect mode of the terminal. When write protect mode is on, all the characters written are in protect mode (background). Characters written this way cannot be written over or cleared (except by clear all) when the terminal is in protect mode (see ptct_crt). The first argument is the output file descriptor. The second argument is a flag, indicating whether the terminal is in write protect mode after the call. If it is nonzero, the terminal is placed in write protect mode.

RETURNS

none

QMOVE QMOVE

NAME

qmove -- move from one string into another

SYNTAX

```
qmove(ptrl,ptr2,length)
char *ptrl,*ptr2;
int length;
```

DESCRIPTION

This function moves characters from the first array to the second array for length.

RETURNS

none

SCOMP SCOMP

NAME

scomp -- compare two strings

SYNTAX

```
scomp(str1,str2,len)
char *str1,*str2;
int len;
```

DESCRIPTION

This function compares two strings and returns a value indicating whether they are the same. If they are not the same, it returns a value indicating which one is before the other in standard ascending order. Len is the number of characters to compare. If len is Ø, this function assumes that both strings are null terminated and compares them accordingly.

RETURNS

```
<Ø -- str1 is before str2
 Ø -- they are the same
>Ø -- str2 is before str1
```

SETCOOK SETCOOK

NAME

setcook -- set the terminal in cooked mode

SYNTAX

setcook()

DESCRIPTION

This function sets the terminal (standard output) into cooked mode. (Refer to the function stty(1) in the TRS-XENIX System Reference Manual.)

RETURNS

none

SETRAW **SETRAW**

NAME

setraw -- set the terminal into raw mode

SYNTAX

setraw()

DESCRIPTION

This function sets the terminal (standard output) into raw mode. (Refer to the function stty(1) in the TRS-XENIX System Reference Manual.)

RETURNS

none

STRCMP STRCMP

NAME

strcmp -- compare two strings

SYNTAX

```
strcmp(str1,str2)
char *str1,*str2;
```

DESCRIPTION

This function compares two strings and returns a value indicating whether they are the same. If they are not the same, the function returns a value indicating which one is before the other in standard ascending order. Both strings should be null terminated.

RETURNS

```
<Ø -- str1 is before str2
 Ø -- they are the same
>Ø -- str2 is before str1
```

UNLOCK **UNLOCK**

NAME

unlock -- unlock a resource reserved by lock

SYNTAX

```
unlock(c)
char c;
```

DESCRIPTION

Use this function to free a resource reserved by the function lock. The same letter code is used to designate the resource. The result is to remove the file named LOCKx, where x is the character used to designate the resource. The file is either located in the user's working directory or $DBPATH, if this environment variable is set.

RETURNS

```
 Ø -- the resource is freed
-1 -- the resource was not locked
```

SEE ALSO

lock

VALCHAR VALCHAR

NAME

valchar -- validate a character against a set of characters

SYNTAX

```
valchar(c,str)
char c, *str;
```

DESCRIPTION

Use this function to verify that a character is a member of a specified set of characters. The first argument is the character to be validated, and the second argument is a string containing all of the valid characters. The string should be null terminated.

RETURNS

Ø -- the character is not a member of the set
1 -- the character is a member of the set

VALSTR **VALSTR**

NAME

valstr -- validate a string against a set of strings

SYNTAX

```
valstr(strng,set)
char *strng,**set;
```

DESCRIPTION

This function verifies that a passed string is a member of a set of passed strings. The first argument is the string to be validated. The second argument is an array containing all valid sets of strings. All strings should be null terminated. The set of strings should be terminated by a null pointer.

RETURNS

Ø -- the string didn't match any of the set
1 -- the string matched one of the set

APPENDIX A
THE UNIFY RM/COBOL INTERFACE

DISREGARD THIS APPENDIX UNLESS OPERATING WITH THE TRS-XENIX COBOL DEVELOPMENT SYSTEM.

This section describes the calling sequences and procedures used to interface RM/COBOL programs to UNIFY. UNIFY uses the "CALL USING" syntax to invoke the various database functions. Not all of the functions listed in the UNIFY Reference Manual are available to COBOL programs. Generally, only the high level routines which are not "C" language specific are part of the interface.

Three functions which are not part of the "C" interface are available to COBOL programmers. They are SETREC, READREC, and WRITEREC ("Record Level I/O Functions" in this Appendix). These functions let programmer defined "records" be read and written in one operation instead of a field at a time.

Data Type Compatibility

COBOL makes extensive use of binary coded decimal to represent numeric fields. Since UNIFY doesn't have this data type, conversions must be performed to support COBOL operations. The following is the mapping between UNIFY data descriptions and the COBOL data types which will hold them.

UNIFY	LENGTH	COBOL (USAGE IS)	EXAMPLE PICTURE*	FORMAT
STRING		DISPLAY	X(<L>)	
NUMERIC	1-4	COMP-1	S9(<L>)	
NUMERIC	5-9	COMP-3	S9(<L>)	
AMOUNT		COMP-3	S9(<L>)V99	
DATE		COMP-3	S9(6)	MMDDYY
TIME		COMP-3	S9(4)	HHMM
FLOAT		(NONE)		

*All numeric fields must be signed. "<L>" represents the UNIFY Schema length entry for the database field.

In general, these conversions are performed by the UNIFY routines. The programmer does not need to worry about them, except when establishing the buffers UNIFY will read and write.

No implicit or explicit FILLER can exist between the fields in a "view" or between fields that make up a COMB field. When accessing any such group of fields, UNIFY must not encounter any "slack" bytes. Note that implicit FILLER often exists between data items at Ø1 or 77 levels to cause them to begin at natural address boundaries (half-word boundaries, word boundaries, and so forth). The fields that make up views and COMB fields should be declared as elementary items within records.

Files to "Copy" into a Program

Several files must be copied into a source program, either manually or via the COPY statement, in order to use the interface. The first file, UCOBOL.H, consists of the Working-Storage Section entries, which associate a number with each UNIFY function. The other copy files also consist of Working-Storage Section entries. They can be created using the conversion program HFILECC described in the next section.

File.h contains the UNIFY record and field names and their associated numbers (as shown in the SCHEMA REPORTS). These numbers remain constant for the life of the database; therefore, recompilation is not necessary when the database design changes. If the screen handler is going to be used, the <screen>.h file for each screen must also be converted.

PROGRAM HFILECC

Program HFILECC is available to convert the UNIFY .h files (described in the previous section) into Working-Storage Section entries. It reads the UNIFY file given as the argument and writes the resulting output on the standard output device. In the following example, the output is redirected to a temporary file called temp.

```
HFILECC file.h > temp
```

Program HFILECC ignores input lines that cannot be converted. It attempts to convert the field, record, and screen-field names into COBOL data-names by writing all alphabetic characters in uppercase and by replacing each underscore character with a hyphen. If the name needs further editing, the message "EDIT" is written in columns 73 - 80. The associated number must be unsigned. Only the left four digits are written.

HFILECC doesn't determine if the data-name created is a COBOL reserved word. If any reserved words exist in the

resulting file, they can be changed or the entry can be removed if unnecessary.

The Calling Sequence

The following syntax is followed in calling UNIFY routines from COBOL.

```
CALL UNIFY
USING function-name [,status] [,argument-1] ... .
```

The name "UNIFY" is a COBOL data-name declared as follows:

```
Ø1 UNIFY PIC X(5) USAGE IS DISPLAY VALUE IS "UNIFY".
```

Function-name is the name of the UNIFY function the programmer wishes to invoke.

Status is a COMP-1 variable which receives the function's return value. Such a variable is not used if there are no returns from the function being invoked. The arguments generally coincide with the arguments documented in the UNIFY C Language Interface Chapter for the function involved. Exceptions are covered later in this Appendix. The following example shows a call to ADDREC.

```
CALL UNIFY:
     USING ADDREC, ADD-STATUS, CUSTOMER, CUSTOMER-KEY.
```

This call adds a CUSTOMER record to the database with the key contained in CUSTOMER-KEY. The status of the add request is returned in ADD-STATUS. This status corresponds to the returns in the addrec documentation in Chapter 9.

Since CALL UNIFY statements are used with varying arguments, they often cause the compiler to generate warning messages. The messages are unfortunate, but unavoidable.

Record Level I/O Functions

Gfield and pfield do not appear in the COBOL interface. Instead, they have been replaced by three "record" level functions. The first function is SETREC. This function establishes a set of fields which are all read or written at the same time. All fields do not have to be in the UNIFY record type. This function gives the programmer the opportunity to create "views". SETREC uses a buffer which will be subsequently read and written by the UNIFY calls. The variables in this buffer must conform in type and sequence to the fields forming the SETREC view. The following syntax is used in calling SETREC.

```
CALL UNIFY
     USING SETREC, view, buffer, field-list, error-subscript.
```

View is a COMP-1 variable which is assigned a number by SETREC. If the number is non-negative, it identifies the view (of the listed fields) and must be used when calling READREC and WRITEREC. A negative view value indicates an error condition; therefore, test it as you would a status value. The possible negative values and their meanings are discussed below. Buffer is the name of the variable which is used for reading and writing the view. The field list is an array of COMP-1 variables containing a zero terminated list of the fields in the view. None of the fields may be of type COMB.

These are the possible negative view values and their meanings:

- -1 - An invalid field was encountered in the field list.
- -2 - A field's type is invalid.
- -3 - No fields in the list.
- -4 - UNIFY's array of field lists is full.

If view contains -1 or -2, the last argument, error-subscript, contains the subscript of the offending field in the field list. Error-subscript must be a COMP-1 variable.

The other two functions are READREC and WRITEREC. These functions read and write data to and from a UNIFY database. The calling sequences are as follows:

```
CALL UNIFY
     USING READREC, status, view, error-subscript.

CALL UNIFY
     USING WRITEREC, status, view, error-subscript.
```

View is a COMP-1 variable that contains the non-negative view identifier obtained by a call to SETREC. A zero status indicates the call was successful. Listed below are the other values that status may contain after these functions are called.

For READREC:

- -1 - Read access for field not allowed.
- -9 - View identifier number is not known.

For WRITEREC:

- -1 - Attempt to change the key of a record which currently has other records referencing it.
- -2 - Attempt to store a duplicate key.
- -3 - Attempt to form an explicit relationship to a record which doesn't exist.
- -4 - Write access for field not allowed.
- -9 - View identifier number not known.

A field list subscript is placed in error-subscript for all status values listed other than -9. If any of the record types accessed by the view are not current when either one of these functions is executed, the program is aborted.

Other UNIFY Functions

This section summarizes the other UNIFY functions available from COBOL. It contains the calling sequences for the available functions and documentation unique to the COBOL interface. Refer to Chapter 9 for function documentation and to "The Calling Sequence" in this appendix for a review of the calling sequence syntax.

All data elements used to pass or receive integer arguments must be declared as COMP-1, unless otherwise stated. Also, when a null-terminated character string is required, it must be explicitly declared as a string (USAGE IS DISPLAY) followed by at least one byte of binary zeros (refer to "Example Program" in this Appendix).

RECORD FUNCTIONS

```
CALL UNIFY
     USING UACCESS, status, rnum, key.
```

(see ACCESS, Chapter 9)

```
CALL UNIFY
     USING ADDREC, status, rnum, key.

CALL UNIFY
     USING UDELETE, status, rnum.
```

(see DELETE, Chapter 9)

```
CALL UNIFY
     USING LOC, rnum, addr.
```

addr -- This argument will receive a 4 byte binary value. The chosen COBOL

data-name's PICTURE character-string
should be X(4).

See also: SETLOC

```
CALL UNIFY
     USING LOCKREC, status, rnum.

CALL UNIFY
     USING SEQACC, status, rnum, direction.

CALL UNIFY
     USING SETLOC, status, rnum, loc.
```

loc -- See function LOC, argument addr.

```
CALL UNIFY
     USING ULOCKREC, rnum.
```

SELECTION PROCESSOR FUNCTIONS

```
CALL UNIFY
     USING CLOSESF, sf.
```

sf -- see function OPENSF.

```
CALL UNIFY
     USING CLRSITM, field, rfield.

CALL UNIFY
     USING ENTSITM, status, field, val1, val2,
                    mode, range-flag.
```

range-flag -- Set this COMP-1 variable to 1 if the selection is a range (val1 and val2 are field occurrences that define the

range), otherwise set it to zero and repeat the val1 argument in the val2 position.

```
CALL UNIFY
     USING FRSTSEL, status, sf.
```

sf -- See function OPENSF.

```
CALL UNIFY
     USING MTCHITM, status, selfile, field, rfield.

CALL UNIFY
     USING NEXTSEL, status, sf.
```

sf -- See function OPENSF.

```
CALL UNIFY
     USING OPENSF, status, selfile, sf.
```

status -- These are the possible status values and their meanings.

- Ø - Open was successful.
- -1 - Selfile could not be opened.
- -2 - Selfile is not in the correct format.
- -3 - The maximum number of open selection files has been reached.
- -4 - An error due to "sbrk".

sf - This argument will receive a memory address needed only by other UNIFY functions. The chosen COBOL data-name's PICTURE character-string should be X(6). This is more than needed, but it's better than not enough.

See also: CLOSESF, FRSTSEL, NEXTSEL, PREVSEL.

```
CALL UNIFY
     USING PREVSEL, status, sf.
```

sf -- See function OPENSF.

```
CALL UNIFY
     USING SELSORT, status, rnum, fldtbl.
```

fldtbl -- This must be the name of an array of COMP-1 variables. The array must be loaded with the desired list of elements and the -1 terminator, as described in the unisort documentation in Chapter 9. The list cannot contain more than 48 elements.

```
CALL UNIFY
     USING UNISEL, status, selfile, xrec, count.
```

count - Must be a COMP-3 variable with PICTURE character-string S9(9).

EXPLICIT RELATIONSHIP FUNCTIONS

```
CALL UNIFY
     USING FACCESS, status, rnum, field.

CALL UNIFY
     USING SNEXTREC, status, field, rfield.

CALL UNIFY
     USING SPREVREC, status, field, rfield.
```

SECONDARY INDEX FUNCTIONS

```
CALL UNIFY
     USING BTNEXT, status, field.

CALL UNIFY
     USING BTSRCH, status field, buf.

CALL UNIFY
     USING CLOSBT, status, field.

CALL UNIFY
     USING OPNBTS, status, field.
```

TERMINAL I/O FUNCTIONS

```
CALL UNIFY
     USING CLEANCRT.

CALL UNIFY
     USING CLEARSCR.

CALL UNIFY
     USING DSPLY, ssfld, esfld.

CALL UNIFY
     USING ERASPRMP, ssfld, esfld.

CALL UNIFY
     USING INBUF, status, sfld, buf.

CALL UNIFY
     USING UINPUT, status, sfld.
```

(see INPUT, Chapter 9)

```
CALL UNIFY
     USING LOADSCR, scrn.
```

```
CALL UNIFY
     USING OUTBUF, sfld, buf.

CALL UNIFY
     USING UOUTPUT, sfld.
```

(see OUTPUT, Chapter 9)

```
CALL UNIFY
     USING PRMP, x, y, str.

CALL UNIFY
     USING PRTMSG, x, y, strng.

CALL UNIFY
     USING YORN, status, str.
```

UTILITY FUNCTIONS

```
CALL UNIFY
     USING ULOCK, c.
```

(see LOCK, Chapter 9)

```
CALL UNIFY
     USING PRIAMD, status, line.

CALL UNIFY
     USING UUNLOCK, status, c.
```

(see UNLOCK, Chapter 9)

Installation Instructions

A shell script called runcobol.ld (see UNIFY's bin directory) is used to load the UNIFY/COBOL interface with RM/COBOL's file: runcobol.o. As written, runcobol.ld produces /bin/runcobol. This means that the current /bin/runcobol will be destroyed if it is not moved.

Use of the new /bin/runcobol is no different than use of the current one. Runcobol.ld expects to find runcobol.o and file cobc.a in the /usr/lib directory; however, it can be customized if necessary.

In short, these steps should be followed:

1) Move or backup the current /bin/runcobol.

2) Customize runcobol.ld, if necessary.

3) Run runcobol.ld.

Note: This interface may not run properly with RM/COBOL releases prior to 1.6. If a memory fault occurs when the COBOL program using UNIFY should terminate, this may be the problem.

Example Program

This section includes an example program preceded by a discussion of the related schema and COPY files.

The example program is intended for use as a reference. Several relevant declarations and data structures are shown, and a good representation of the interface functions are used. Although the program performs fairly meaningful actions, it is not intended to be keyed in and used as a working example.

The schema is simply the tutorial schema with one record type added. Only that record (cobcr) and the model record from the tutorial schema are relevant to the example program. The model record is used because of its COMB key, and record cobcr is used because it contains several data types.

Following are the schema listings of the two records.

```
model
    *mokey                      COMB
       monum                    NUMERIC    7
       momano       mano        NUMERIC    4
    mdes                        STRING    3Ø

cobcr
    *intf                       NUMERIC    4
    longf                       NUMERIC    9
    datef                       DATE
    amtf                        AMOUNT     7
    strngf                      STRING    19
    hamtf                       AMOUNT    1Ø
    hrf                         TIME
```

The following is the complete listing of the schema's file.h. Following file.h is a copy of a file called FILE.H. Program HFILECC is used to create FILE.H from file.h. Then FILE.H is edited to remove some unneeded lines. Note that HFILECC does not check for COBOL reserved words.

```
$define first 1       /* file.h */
$define next 2
$define only 3
$define prev 4
$define last 5
$define FN 1
$define FS 2
$define RN 4
$define RS 8
$define mano 1
$define mname 2
$define madd 3
$define mokey 4
$define monum 5
$define momano 6
$define mdes 7
$define sno 8
$define imodel 9
$define imodel_monum 1Ø
$define imodel_momano 11
$define iad 12
$define isal 13
$define mcity 14
$define mstate 15
$define mzip 16
$define iorder 17
$define ipamt 18
$define cnum 19
$define cname 2Ø
$define caddr 21
$define ccity 22
$define cstate 23
$define czip 24
$define cphone 25
$define onum 26
$define odate 27
$define ocust 28
$define intf 29
$define longf 3Ø
$define datef 31
$define amtf 32
$define strngf 33
```

```
$define hamtf 34
$define hrf 35
$define manf 1
$define model 2
$define item 3
$define customer 4
$define order 5
$define cobcr 6
```

This is the listing of the COPY file: FILE.H.

```
77  MOKEY
        PIC 9999  USAGE COMP-1  VALUE 4.
77  MONUM
        PIC 9999  USAGE COMP-1  VALUE 5.
77  MOMANO
        PIC 9999  USAGE COMP-1  VALUE 6.
77  MDES
        PIC 9999  USAGE COMP-1  VALUE 7.
77  INTF
        PIC 9999  USAGE COMP-1  VALUE 29.
77  LONGF
        PIC 9999  USAGE COMP-1  VALUE 3Ø.
77  DATEF
        PIC 9999  USAGE COMP-1  VALUE 31.
77  AMTF
        PIC 9999  USAGE COMP-1  VALUE 32.
77  STRNGF
        PIC 9999  USAGE COMP-1  VALUE 33.
77  HAMTF
        PIC 9999  USAGE COMP-1  VALUE 34.
77  HRF
        PIC 9999  USAGE COMP-1  VALUE 35.
77  MODEL
        PIC 9999  USAGE COMP-1  VALUE 2.
77  COBCR
        PIC 9999  USAGE COMP-1  VALUE 6.
```

The UNIFY Screen sgetkys is used by the example program to allow a user to enter record key values (see paragraphs GET-KEYS and GET-KEYS-1). The screen's fields are associated with schema fields: intf, monum, and momano. Program HFILECC uses file sgetkys.h to create COPY file SGETKYS.H. These two files are as follows:

```
#define sintf       Ø       /* sgetkys.h */
#define smonum      1
#define smomano     2
```

```
77  SINTF
        PIC 9999  USAGE COMP-1  VALUE Ø.
77  SMONUM
        PIC 9999  USAGE COMP-1  VALUE 1.
77  SMOMANO
        PIC 9999  USAGE COMP-1  VALUE 2.
```

COPY file UCOBOL.SML is an edited version of file UCOBOL.H. It is very important to save a copy of UCOBOL.H. Portions of it are needed by every program that calls the UNIFY interface.

This is COPY file UCOBOL.SML:

```
77  UACCESS
        PIC 9999  USAGE COMP-1  VALUE Ø.
77  ADDREC
        PIC 9999  USAGE COMP-1  VALUE 1.
77  CLEANCRT
        PIC 9999  USAGE COMP-1  VALUE 4.
77  CLOSESF
        PIC 9999  USAGE COMP-1  VALUE 7.
77  UDELETE
        PIC 9999  USAGE COMP-1  VALUE 9.
77  ENTSITM
        PIC 9999  USAGE COMP-1  VALUE 11.
77  FRSTSEL
        PIC 9999  USAGE COMP-1  VALUE 14.
77  INBUF
        PIC 9999  USAGE COMP-1  VALUE 15.
77  LOADSCR
        PIC 9999  USAGE COMP-1  VALUE 17.
77  NEXTSEL
        PIC 9999  USAGE COMP-1  VALUE 22.
77  OPENSF
        PIC 9999  USAGE COMP-1  VALUE 23.
77  READREC
        PIC 9999  USAGE COMP-1  VALUE 31.
77  SETREC
        PIC 9999  USAGE COMP-1  VALUE 35.
77  UNISEL
        PIC 9999  USAGE COMP-1  VALUE 39.
77  WRITEREC
        PIC 9999  USAGE COMP-1  VALUE 41.
77  YORN
        PIC 9999  USAGE COMP-1  VALUE 42.
```

The values for EQ-CODE and GT-CODE in the example program are obtained by inspecting file unisel.h. File unisel.h is an include file shown in the documentation of function entsitm in Chapter 9. Program HFILECC is not designed to handle negative

numbers; therefore, the entry for SS in unisel.h is not converted.

The remainder of this section consists of a listing of the example program: USEUNI.

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID.    USEUNI.
  AUTHOR.   WAM.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER.          YOURS.
  OBJECT-COMPUTER.          YOURS.

  DATA DIVISION.
  WORKING-STORAGE SECTION.
  COPY "../../include/UCOBOL.SML".

  COPY "../../def/FILE.H".

  COPY "../../def/SGETKYS.H".

*** CINTF will hold COBCR'S key *

  77  CINTF                  PIC S9(4) USAGE COMP-1.
  77  CINTF-SAVE             PIC S9(4) USAGE COMP-1.
  77  VIEW-NUM               PIC S9(4) USAGE COMP-1.
  77  ERR-SUBSCRIPT          PIC S9(4) USAGE COMP-1.
  77  STATUS-VAL             PIC S9(4) USAGE COMP-1.
  77  ACPT-CHAR              PIC X.
  77  R-FLAG                 PIC S9 USAGE COMP-1.
  77  DVAL-1                 PIC S9(6) USAGE COMP-3.
  77  DVAL-2                 PIC S9(6) USAGE COMP-3.
  77  MODE-NR                PIC S9 USAGE COMP-1.
  77  HVAL                   PIC S9(1Ø)V99 USAGE COMP-3.
  77  COUNT-VAL              PIC S9(9) USAGE COMP-3.
  77  SF-ADDR                PIC X(6).

  Ø1  UNIFY                   PIC X(5) USAGE DISPLAY;
                              VALUE "UNIFY".

*** VIEW1 is a buffer for SETREC, READREC, WRITEREC *

  Ø1  VIEW1.
      Ø2  CSTRNGF             PIC X(19) USAGE DISPLAY.
      Ø2  CHRF                PIC S9(4) USAGE COMP-3.
      Ø2  CMDES               PIC X(3Ø) USAGE DISPLAY.
      Ø2  CDATEF              PIC S9(6) USAGE COMP-3.
```

```
      02  CLONGF             PIC S9(9) USAGE COMP-3.
      02  CAMTF              PIC S9(7)V99 USAGE COMP-3.
      02  CHAMTF             PIC S9(10)V99 USAGE COMP-3.
  01  CHRF-O                 PIC ----9.
  01  CDATEF-O               PIC Z9/99/99.
  01  CLONGF-O               PIC -(9)9.
  01  CAMTF-O                PIC --,---,--9.99.
  01  CHAMTF-O               PIC --,---,---,--9.99.
  01  COUNT-VAL-O            PIC ----,---,--9.

*** CMOKEY will hold MODEL'S record key *

  01  CMOKEY.
      02  CMONUM             PIC S9(7) USAGE COMP-3.
      02  CMOMANO            PIC S9(4) USAGE COMP-1.

  01  CMOKEY-SAVE.
      02  CMONUM-S           PIC S9(7) USAGE COMP-3.
      02  CMOMANO-S          PIC S9(4) USAGE COMP-1.

  01  VIEW-ARY.
      02  V-FIELD            PIC S9(4) USAGE COMP-1;
                             OCCURS 10.
  01  SFILE-NM.
      02  SFILE-S            PIC X(5) USAGE DISPLAY
                             VALUE "cobcr".
      02  SFILE-T            PIC 9 USAGE COMP-1 VALUE 0.

*** The following code values are from unisel.h *

  01  GT-CODE                PIC S9 USAGE COMP-1 VALUE 5.
  01  EQ-CODE                PIC S9 USAGE COMP-1 VALUE 8.

  01  REC-STATS-M.
      02  REC-STATS-MSG      PIC X(24) USAGE DISPLAY
                             VALUE SPACES.
      02  REC-STATUS-O       PIC ----9.
      02  FILLER             PIC X(14)
                             VALUE ", AND ERR-S = ".
      02  ERR-SUBSCRIPT-O    PIC ----9 VALUE ZERO.

  01  DISPLAY-STATS-M.
```

```
       Ø2  STATS-MSG          PIC X(24) USAGE DISPLAY
                              VALUE SPACES.
       Ø2  STATUS-VAL-O       PIC ----9.

   Ø1  GETKYS-SCRN.
       Ø2  SGETKYS-S          PIC X(7) USAGE DISPLAY;
                              VALUE "sgetkys".
       Ø2  SGETKYS-T          PIC 9 USAGE COMP-1 VALUE Ø.

   Ø1  CONTINUE-MSG.
       Ø2  CONTINUE-S         PIC X(9) USAGE DISPLAY;
                              VALUE "CONTINUE?".
       Ø2  CONTINUE-T         PIC 9 USAGE COMP-1 VALUE Ø.

PROCEDURE DIVISION.

CONTROL-SECTION.
    PERFORM SETREC-CALL.
    PERFORM GET-KEYS THRU GET-KEYS-1.
    PERFORM UACCESS-CALL.
    PERFORM READREC-CALL.

    MOVE CINTF TO CINTF-SAVE.
    MOVE CMOKEY TO CMOKEY-SAVE.

    PERFORM GET-KEYS THRU GET-KEYS-1.
    PERFORM ADDREC-CALL.
    PERFORM WRITEREC-CALL.

    MOVE CINTF-SAVE TO CINTF.
    MOVE CMOKEY-SAVE TO CMOKEY.

    PERFORM UACCESS-CALL.
    PERFORM UDELETE-CALL.

    PERFORM ENTSITM-CALLS.
    PERFORM UNISEL-CALL.
    PERFORM OPENSF-CALL.
    PERFORM FRSTSEL-CALL.
    PERFORM READREC-CALL.
    PERFORM NEXTSEL-CALL.
    PERFORM READREC-CALL.
```

```
    PERFORM CLOSESF-CALL.

STOP-RUN.
    STOP RUN.


SETREC-CALL.
    MOVE STRNGF TO V-FIELD (1).
    MOVE HRF    TO V-FIELD (2).
    MOVE MDES   TO V-FIELD (3).
    MOVE DATEF  TO V-FIELD (4).
    MOVE LONGF  TO V-FIELD (5).
    MOVE AMTF   TO V-FIELD (6).
    MOVE HAMTF  TO V-FIELD (7).
    MOVE ZERO   TO V-FIELD (8).

    CALL UNIFY
        USING SETREC, VIEW-NUM, VIEW1,
              VIEW-ARY, ERR-SUBSCRIPT.
    MOVE VIEW-NUM TO STATUS-VAL.
    MOVE "FROM SETREC, VIEW NR. = " TO REC-STATS-MSG.
    PERFORM DISPLAY-REC-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.
GET-KEYS
    DISPLAY "[sgetkys]", LINE 1, POSITION 1, ERASE.
    CALL UNIFY USING LOADSCR, GETKYS-SCRN.
GET-KEYS-1.
    MOVE -1 TO STATUS-VAL.
    PERFORM INBUF-SINTF UNTIL STATUS-VAL = Ø.
    MOVE -1 TO STATUS-VAL.
    PERFORM INBUF-SMONUM UNTIL STATUS-VAL = Ø.
    MOVE -1 TO STATUS-VAL.
    PERFORM INBUF-SMOMANO UNTIL STATUS-VAL = Ø.
    CALL UNIFY USING YORN, STATUS-VAL, CONTINUE-MSG.
    IF STATUS-VAL = 1,
        NEXT SENTENCE;
    ELSE CALL UNIFY USING CLEANCRT;
        GO TO GET-KEYS-1.
INBUF-SINTF.
    CALL UNIFY USING INBUF, STATUS-VAL, SINTF, CINTF.
INBUF-SMONUM.
    CALL UNIFY USING INBUF, STATUS-VAL, SMONUM, CMONUM.
```

```
INBUF-SMOMANO.
    CALL UNIFY USING INBUF, STATUS-VAL, SMOMANO, CMOMANO.

UACCESS-CALL.
    CALL UNIFY USING UACCESS, STATUS-VAL, COBCR, CINTF.
    MOVE "UACCESS COBCR, STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.
    CALL UNIFY USING UACCESS, STATUS-VAL, MODEL, CMOKEY.
    MOVE "UACCESS MODEL, STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

READREC-CALL.
    CALL UNIFY USING READREC, STATUS-VAL,
                VIEW-NUM, ERR-SUBSCRIPT.
    MOVE "FROM READREC,  STATUS = " TO REC-STATS-MSG.
    PERFORM DISPLAY-REC-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.
    DISPLAY "CONTENTS OF THE VIEW:", LINE 9, POSITION 1.
    DISPLAY CSTRNGF, LINE Ø, POSITION 5.
    MOVE CHRF TO CHRF-Ø.
    DISPLAY CHRF-Ø, LINE Ø, POSITION 5.
    DISPLAY CMDES, LINE Ø, POSITION 5.
    MOVE CDATEF TO CDATEF-Ø.
    DISPLAY CDATEF-Ø, LINE Ø, POSITION 5.
    MOVE CLONGF TO CLONGF-Ø.
    DISPLAY CLONGF-Ø, LINE Ø, POSITION 5.
    MOVE CAMTF TO CAMTF-Ø.
    DISPLAY CAMTF-Ø, LINE Ø, POSITION 5.
    MOVE CHAMTF TO CHAMTF-Ø.
    DISPLAY CHAMTF-Ø, LINE Ø, POSITION 5.
    PERFORM PAUSE

ADDREC-CALL.
    CALL UNIFY USING ADDREC, STATUS-VAL, COBCR, CINTF.
    MOVE " ADDREC COBCR, STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.
    CALL UNIFY USING ADDREC, STATUS-VAL, MODEL, CMOKEY.
    MOVE " ADDREC MODEL, STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
```

```
    IF STATUS-VAL < Ø GO TO STOP-RUN.

WRITEREC-CALL.
    CALL UNIFY USING WRITEREC, STATUS-VAL,
                VIEW-NUM, ERR-SUBSCRIPT.
    MOVE "FROM WRITEREC, STATUS = " TO REC-STATS-MSG.
    PERFORM DISPLAY-REC-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

UDELETE-CALL.
    CALL UNIFY USING UDELETE, STATUS-VAL, COBCR.
    MOVE "UDELETE COBCR, STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

ENTSITM-CALLS.
    MOVE 1Ø182 TO DVAL-1.
    MOVE 123183 TO DVAL-2.
    MOVE 1 TO R-FLAG.
    CALL UNIFY USING ENTSITM, STATUS-VAL, DATEF,
                DVAL-1, DVAL-2, EQ-CODE, R-FLAG.
    MOVE "ENTSITM DATEF, STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.
    MOVE Ø TO R-FLAG.
    MOVE 5ØØØØØØ.ØØ TO HVAL.
    CALL UNIFY USING ENTSITM, STATUS-VAL, HAMTF,
                HVAL, HVAL, GT-CODE, RFLAG.
    MOVE "ENTSITM HAMTF, STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

UNISEL-CALL
    MOVE Ø TO COUNT-VAL.
    CALL UNIFY USING UNISEL, STATUS-VAL, SFILE-NM,
                COBCR, COUNT-VAL.
    MOVE "UNISEL CALL,   STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    DISPLAY "COUNT AFTER UNISEL IS: ",
        LINE 4, POSITION 1, ERASE.
    MOVE COUNT-VAL TO COUNT-VAL-Ø.
    DISPLAY COUNT-VAL-Ø, LINE Ø, POSITION Ø.
```

```
    PERFORM PAUSE.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

OPENSF-CALL.
    IF  COUNT-VAL < 1,
        MOVE -9 TO STATUS-VAL;
    ELSE CALL UNIFY USING OPENSF,
        STATUS-VAL, SFILE-NM, SF-ADDR.
    MOVE "OPENSF CALL,   STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

FRSTSEL-CALL.
    CALL UNIFY USING FRSTSEL, STATUS-VAL, SF-ADDR.
    MOVE "FRSTSEL CALL,  STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

NEXTSEL-CALL.
    CALL UNIFY USING NEXTSEL, STATUS-VAL, SF-ADDR.
    MOVE "NEXTSEL CALL,  STATUS = " TO STATS-MSG.
    PERFORM DISPLAY-STATS.
    IF STATUS-VAL < Ø GO TO STOP-RUN.

CLOSESF-CALL.
    CALL UNIFY USING CLOSESF, SF-ADDR.

DISPLAY-REC-STATS.
    MOVE STATUS-VAL TO REC-STATUS-Ø.
    MOVE ERR-SUBSCRIPT TO ERR-SUBSCRIPT-Ø.
    DISPLAY REC-STATS-M, ERASE, LINE 5, POSITION 1.
    PERFORM PAUSE.

DISPLAY-STATS.
    MOVE STATUS-VAL TO STATS-VAL-Ø.
    DISPLAY DISPLAY-STATS-M, ERASE, LINE 5, POSITION 1.
    PERFORM PAUSE.

PAUSE.
    DISPLAY "PRESS <ENTER> TO CONTINUE. ",
        BEEP, LINE 7, POSITION 1.
    ACCEPT ACPT-CHAR, LINE Ø, POSITION Ø.
```

```
END PROGRAM.
```

APPENDIX B
VI EDITOR COMMANDS

The following abbreviations are used in this documentation:

<psn> -- refers to a position up to which you wish to affect and may refer to the following:

<cmd> <w> -.....up to the next word (e.g. <d><w> means to delete up to the next word)
<cmd> <$> - .....up to the end of the current line
<cmd> <^> - .....[back] to the non-blank start of the line
<cmd> <cmd> - .....if you repeat the previous command, the whole line on which the cursor is located is affected (e.g. <d><d>)
<cmd> <e> - .....up to the next word, leaving the punctuation
<cmd> <b> - .....the preceding word
<cmd> <)> - .....the rest of the current sentence
<cmd> <}> - .....the rest of the current paragraph
<cmd> <]> <]>.....the rest of the current section
<cmd> <G> - .....up to the end of the file
<cmd> <H> - .....to the top of the screen
<cmd> <L> - .....up to the last line of the screen
<cmd> <$> - .....up to the end of the current line
<cmd> <f> c - .....up to (and including) the specified character
<cmd> <t> c - .....up to (but not including) the specified character

---

lno -- refers to a line number, with the following substitutions allowed:

<.> for current line number
<$> for last line number (end of file)
<.> <+> number for next number of lines

---

num -- refers to a repetition number; if omitted, one is usually assumed. If the default number is other than one, this is stated.

---

pat -- refers to a pattern or string. If you wish to use "reserved characters," such as </>, <$>, <,>, etc, then

precede that character with a backslash (<ctrl><9>).

Entering vi

To enter vi from SQL, type **edit** after the SQL (>) prompt.

Use the following commands to enter vi from the Unify shell:

vi file.ext.....edits the file named file.ext
vi +number file.ext.....edit the file beginning at the specified line number
vi + file.ext.....edit the file beginning at the last line
vi -t tag.....starts with a tag for which you are searching
vi -r.....list saved files
vi -r file.....edit a file recovered when the system crashes
vi +/ pat file.ext.....edit the file beginning with a search for string
vi file1 file2.....edit files consecutively (via <:> <n>)

Cursor Movements

(Any of these keys can be preceded by a number for repeat features.)

<k> or <ctrl>-<P>.....move cursor to character above the cursor
<l>.....advance cursor one position to the right
<h>.....move cursor one character to the left (backspace)
<j> or <ctrl>-<N>.....move cursor to character below cursor
<+> or <ENTER>.....move cursor to first non-blank character in next line
<->.....move cursor to first non-blank character in previous line

```
<Ø>.........................move cursor to the beginning of the
                            current line
<spacebar>..................move cursor one character to right
-----------------------------------
<w>.........move cursor to beginning of next word
<W>.........move cursor to beginning of next whitespace
            delimited word
<b>.........move cursor to beginning of the current word
<B>.........move cursor to beginning of the current word,
            ignoring punctuation (will not ignore beginning
            quotation marks)
<e>.........move cursor to end of current word
<E>.........move cursor forward to the end of a word
-----------------------------------
< ) >.......move cursor to beginning of next sentence
<(>.........move cursor to beginning of the current sentence
<}>.........move cursor to beginning of next paragraph
<{>.........move cursor to beginning of the current paragraph
<]> <]>.....move cursor to beginning of next section or
            function
<[> <[>.....move cursor to beginning of the previous section
            or function
```

Screen Commands

```
<ctrl>-<D>.....scroll down 1/2 screen
number <ctrl>-<D>...scroll down number / 2 screenfuls
<ctrl>-<U>.....scroll up 1/2 screen
number <ctrl>-<U>...scroll up number / 2 screenfuls
<ctrl>-<F>.....move forward one page
<ctrl>-<B> or <F2>..move back one page
<H>............home cursor to top line of the current screen
number <H>.....move cursor to the numbered line of the
               current screen
<M>............move cursor to the middle line of the current
               screen
```

```
<L>.................move cursor to the bottom line of the current
                    screen
number <L>..........move cursor to the specified line from
                    the bottom of the current screen
<">.................move cursor to its prior location before last
                    command
<^> (carat).........move cursor to first non-blank position on
                    line
<$>.................move cursor to end of current line
-------------------------------------
<z> <ENTER>.........make current line top of screen
<z> <.>.............make current line center of screen
<z> <->.............make current line bottom of screen
<z> number <ENTER>..redraw screen with only the "number" of lines
                    and make current line the top line
-------------------------------------
<f> c...............move cursor forward to next specified
                    character in the line
<F> c...............move cursor back to the previously specified
                    character in the line
<t> c...............move cursor forward up to but not on a given
                    character
<T> c...............locate a character before the cursor in the
                    current line and place the cursor directly
                    after that character
<;>.................repeat previous <f>, <F>, <t>, or <T> command
<,> (comma).........inverse of <;>
<%>.................move cursor to matching parens, () {} [] but
                    not <>
<G>.................move cursor to the last line of the file
number <G>..........move cursor to line number specified
</> pat <ENTER>.....move cursor to next place string occurs in
                    file
</> pat </> +n
<ENTER>.............move cursor to nth line after string
</> <^> pat.........move cursor to the next place string occurs
                    at the beginning of a line
</> pat <S>.........move cursor to the next line ending with the
                    string
</> pat </> <z> <->.move next line containing string to bottom
                    of the screen
```

<?> pat.....move cursor to previous place string occurs in file
<?> pat <?> -n.....move cursor to nth line before string occurs
<n>.....move cursor to next occurrence of last string for which you searched
<N>.....repeat the search for the last string in the opposite direction

Changing the File

Delete Mode

<D>.....delete rest of line
<d> <psn>.....delete from current character to position
<"> <a - z> <d>.....move following deletions into one of the 26 buffers

---

<C>.....change rest of line
<c> <psn>.....change from current character to position
<c> <c> or <S> pat..change the whole line on which the cursor is located

---

<x> or <d>
<spacebar>.....delete the character under the cursor
<X>.....delete the character before the cursor
<r> c.....replace the character under the cursor with the new character
<R>.....replace characters
<s> pat.....replace characters letter-for-letter with the string

---

<J>.....join the line below to the line on which the cursor is located

Insert Mode

<a>.....append (insert) text directly after the current cursor position
<A>.....append (insert) text at the end of the line on which the cursor is located

```
<i>.................insert text in the line just before the
                    cursor
<I>.................insert text at the beginning of the line on
                    which the cursor is located
<o>.................open a blank line for insertion onto the line
                    below the cursor
<O>.................open a blank line for insertion onto the line
                    above the cursor
-------------------------------------
During insert mode {entered with a c i o s A C I O R S}

     <ESC>.....................end current insertion or change, and
                               save all changes
     <ENTER>...................start new line (while in "input mode")
     <ctrl>-<D>................backtab over auto-indent
     <ctrl>-<V>................literalize next character (for inserting
                               control characters)
     <ctrl>-<H> or
     <backspace>...............move cursor to the left, erasing the
                               character in the working buffer, but NOT
                               on the screen
     <ctrl>-<W>................backspace and erase a whole word
     <ctrl>-<@>................repeat last insert text
-------------------------------------
<u>.................undo last command
<U>.................restore current line
-------------------------------------
<.> (period)........repeat last command which made a change or
                    deletion (when using the numbered buffers,
                    the number is incremented)
-------------------------------------
<P>.................put copy of previously deleted text before
                    and above the cursor
<p>.................put copy of previously deleted text after and
                    below the cursor
<x> <p>.............transpose the character under the cursor with
                    the character next to it
<"> <1 - 9> <p>.....put the nth previously deleted text back into
                    the program
```

Marks

<m> <a - z>.....set one of 26 "bookmarks" to the current cursor position
<'> <a - z> .....move to the beginning of the line containing the specified mark
<'> <'>.....return to the line before the last absolute cursor motion
<ctrl>-<E> <a - z>..move to the cursor position of the mark specified
<ctrl>-<E>
<ctrl>-<E>.....return to the cursor position before the last absolute motion
<|>.....move to a particular column (<ctrl>-<Ø>)

Set Commands

<:> set all.....display all options
<:><s><e> option....set options described below (no... turns option off):

- autoindent (ai).....automatic indentation of new lines
- autowrite (aw).....automatic write before :ta, :n, etc.
- ignore case (ic)....ignore letter case in searches
- lisp.....({) } commands deal with S-expressions
- list.....show tabs as ^I and end of lines with a dollar sign ($)
- magic.....the characters . [and * are special in scans
- nomesg.....prohibit terminal messages while editing
- number (nu).....display line numbers
- paragraphs (para)...macro names which start paragraphs
- redraw (re).....redraw screen after character in insert mode
- sections (sect).....macro names which start new sections
- shiftwidth (sw).....set width of shift and indent commands

```
showmatch (sm)......show matching "(" and "{" when ending
                    ")" and "}" are typed
slowopen (slow).....postpone display updates during inserts
term................the kind of terminal you are using
                    (smart/dumb)
wrapmargin..........automatic line breaks between words
wrapscan (ws).......restart searches at top of file if not
                    found below
```

Major Text Editing

```
<:> lnol <,> lno2
<d> <ENTER>.............delete all lines from line #1 to line #2
<:> lnol <,> lno2
<m> lno3 <ENTER>........move lines #1 - #2 to line #2
<:> lno <r> file........insert file after line number
<:> lnol <,> lno2
<w> file................write lines #1 - #2 to file
<:> lnol <,> lno2 <s>
</> patl </> pat2.......changes first occurrence of patl to pat2
                        on lines #1 - #2
<:> lnol <,> lno2 <s> </>
patl </> pat2 </> <g>....changes all occurrences
                        Note: reserved characters in patterns
                        may be denoted by using a backslash
                        (<ctrl>-<9>) in front of desired
                        character / . or $
<:> lnol <,> lno2 <f> </>
patl {</> <g>}..........global find
```

Miscellaneous Commands

```
<ESC>..............cancel partially formed commands
<ctrl>-<L>.........clear and redraw (refresh) the screen
<ctrl>-<R>.........retype the screen {eliminating "@" lines}
<ctrl>-<G>.........show file status (name, current line #,
                   total # of lines, etc.)
```

```
<:> <f>.............show current file and line (same as
                    <ctrl>-<G>)
< < >...............left shift
< > >...............right shift
<!>.................filter through command
<:> <v> <i>.........will return to command mode if you press <:>
                    accidently
-----------------------------------
<y> <psn>...........yank a copy of the following object into the
                    "undo" buffer
<y> <y> or <Y>......yank a copy of the current line into the
                    "undo" buffer
number <Y>..........yank the stated number of lines into the
                    "undo" buffer
<"> <a - z>
<y> <psn>...........yank a copy of the following object into one
                    of 26 buffers
                    Note:  <p> and <P> commands are used with <y>
                    to make copies of code
-----------------------------------
<!><G>..............makes the next command cover from current
                    cursor position to E.O.F.
<:> <!> cmd.........perform mentioned system command, then return
<!> <!> cmd.........include contents of the result of the system
                    command in the text
<:> <s> <h>.........temporarily exit to shell (type <ctrl>-<D> to
                    reenter vi)
<:> <t> <a> tag.....to tag file entry "tag"
-----------------------------------
<:> <n>.............edit the next file in the list vi file1 file2
                    file3 etc
<:> <n> file1 file2.specify new file list
<:> <e> file.ext....edit the file named file.ext
<:> <e> <#>.........edit the previously edited file
<:> <e> <!>.........restart editing of the current file
<:> <e> <+>
file.ext............edit the file starting at the bottom
<:> <e> <+>number...edit the file starting at the specified line
                    number
```

```
<:> <w> <ENTER>.....write out the file
<:> <w> file.ext....write out the file to a file named file.ext
<:> <w> <!>
file.ext............overwrite file file.ext
<:> <w> <q>.........write and quit
<:> <q>.............quit
<Z> <Z>.............exit from vi and save all changes
<:> <q> <!>.........exit from vi and discard all changes
```

APPENDIX C
ERROR MESSAGES

Occasionally, you might see one of the error messages listed in this appendix. Each message is explained in order of the program in which it occurs. Shown are the error messages and their probable causes.

MENUH -- Menu Handler Messages

Invalid option
Either this option is not in the data dictionary, or you have no access privilege for it.

Can't open
The help documentation file for this option does not exist or is not readable.

Target unknown
This is an internal error. The target record type for this ENTER screen is not in the data dictionary. The data dictionary file is probably unreadable and should be recovered from a backup.

Can't access mprec
This menu or program name cannot be located in the data dictionary.

Invalid ID
This login ID is not in the data dictionary.

Invalid password
This password does not correspond with the login ID displayed.

Executable Maintenance Messages

Prog/menu already exists
There is already a program or menu in the data dictionary with this name.

Executable already exists
There is already an executable in the data dictionary with this name.

Executable unknown
There is no executable in the data dictionary with this name.

Illegal entry
You cannot use the executable names "ENTER" or "MENU," as these are reserved for the system.

Invalid index for mprec
This is an internal error. The index for one of the programs in the executable is unusable. Reenter this executable with a different name, or recover the data dictionary from a backup.

Screen unknown
The screen form indicated is not present in the data dictionary.

Enter cr, m or d
On a non-empty line, the options are <ENTER>, modify, or delete.

Enter cr or a
On an empty line, the options are <ENTER> or add.

Menu Maintenance Messages

Menu or prog already exists
There is already a menu or program with this name in the data dictionary.

Menu unknown
There is no menu with this name in the data dictionary.

Too many mline entries
This is an internal error. The current menu has too many menu lines (more than 16). Reenter the menu with a different name, or recover the data dictionary from a backup.

A line number must be entered
Each menu line must be given a number. The menu is arranged in sequence by line number.

Line number be 1-16
The line number entered is not from 1 through 16.

Prog/menu unknown
This program, menu, or ENTER screen name is not in the data dictionary.

Enter cr, a, or q
On an empty line, the valid options are <ENTER>, add, or quit.

Enter cr, m, d, or q
On a non-empty line, the valid options are <ENTER>, modify, delete, or quit.

Group Maintenance Messages

Access record already exists
There is already an access privilege record for that program, menu, or ENTER screen in the data dictionary.

Prog/menu unknown
There is no program, menu, or ENTER screen by that name in the data dictionary.

Employee records exist: Group not deleted
There are employee records related to this group, so the

group record could not be deleted. Delete the employees first, then delete the group.

Please enter M, D, or Q
The valid options are modify, delete, or quit.

No more pages
You are currently on the last page of access privileges.

No previous page
You are currently on the first page of access privileges.

Group already exists
There is already a group with this ID in the data dictionary.

Group unknown
There is no group with this ID in the data dictionary.

Entry point must be a menu
The entry point for each group must be a menu (not a program or ENTER screen).

Menu unknown
There is no menu in the data dictionary with this name.

Employee Maintenance Messages

Access record already exists
There is already an access privilege record for this menu, program, or ENTER screen in the data dictionary.

Prog/menu unknown
This program, menu, or ENTER screen is not in the data dictionary.

Please enter M, D, or Q
The valid options are modify, delete, or quit.

No more pages
The current display shows the last page of access privileges.

No previous page
The current display shows the first page of access privileges.

Illegal ID
The employee ID you entered is the ID of the Unify super user. You cannot enter an ordinary employee with the same ID as the super user.

Login already exists
The employee ID you entered already exists in the data dictionary.

Login rec unknown
The employee ID you entered does not exist in the data dictionary.

Entry point must be a menu
Every employee must enter the system at a menu, and you entered either a program or an ENTER screen name.

Group unknown
The group ID you entered does not exist in the data dictionary.

Menu unknown
The menu name you entered does not exist in the data dictionary.

Enter Help Documentation Messages

Can't access m/p record
There is no program, menu, or ENTER screen with this name in the data dictionary.

System Parameter Maintenance Messages

Entry point must be a menu
The system entry point for the Unify super user must be a menu.

Language unknown
The language code must be either EN (for English) or FR (for French).

Menu unknown
This menu is not present in the data dictionary.

Invalid number of blocks
The number of blocks must be greater than Ø.

Print Help Documentation Messages

Can't find file xxxx.n
The documentation file for this program, menu, or ENTER screen does not exist or is not readable, or the help documentation directory itself is not readable.

Can't access m/p
There is no program, menu, or ENTER screen with this name in the data dictionary.

No documentation was found for this menu
You requested all documentation for this menu to be printed, but none was found. Either the documentation does not exist or is not readable, or the help documentation directory is not readable.

Schema Maintenance Messages

Field exists
This field name already exists in the schema. Use a different field name.

Field limit will be exceeded
The maximum number of fields per record type is 256.

Fields can't have record names
The names of records and fields must be distinct. Use a different field name.

Record exists
This record name already exists in the schema. Use a different record name.

Record limit will be exceeded
The maximum number of record types in the database is 256.

Records can't have field names
The names of records and fields must be distinct. Use a different field name.

Cannot delete combination
All the components of a COMB field must be deleted before the COMB field itself can be deleted.

Field is referenced
All fields that reference this field must be deleted before this one can be deleted.

Record is referenced
All references to this record must be deleted before it can be deleted.

Record not deleted
An unexpected error occurred when you were deleting a record. Make sure that all references to this record have been deleted, try deleting all fields individually, and then try deleting the record again.

Enter m, d, num, or q
The valid commands are modify, delete, a line number, or quit.

No more pages
This is the last page of records (or fields).

No previous page
This is the first page of records (or fields).

Field limit exceeded
This is an internal error. This record type has more than 256 fields. Try to delete the record type and then add it again. If this does not work and you cannot recover the data dictionary from a backup, reenter the schema.

Record limit exceeded
This is an internal error. The schema has more than 256 record types. If you cannot recover the data dictionary from a backup, reenter the schema.

Field must be a parent
The entry in the COMB FIELD column must be the name of a field of type COMB in this record type.

Field must ref a key
The entry in the REF column must be the name of the key field in another record type.

Field must ref another rec
The entry in the REF column must be the name of the key field in another record type.

Field unknown
This field name is not in the data dictionary.

Invalid length
The field length must be greater than Ø.

Invalid type
The valid field types are NUMERIC, FLOAT, AMOUNT, STRING, DATE, TIME, and COMB.

Parent must be of the same rec
The entry in the COMB FIELD column must be the name of a field of type COMB in this record type.

Please enter '*' or space
The entry in the KEY column must be either an asterisk (to

indicate that this field is the primary key) or a space (to indicate that it is not).

Enter m, d, f, num, or q
The valid commands are modify, delete, fields, line number, or quit.

Create Data Base and Reconfigure Data Base Messages

Unable to back up data dictionary. Check amount of free space
A write error occurred when attempting to make a copy of unify.db in the database directory. You probably do not have enough space left in the file system, or you might not have write privilege in the directory, so the file could not be created.

Unable to recover data dictionary. Recover file system from backup
A read or write error occurred when trying to restore the backup copy of unify.db. As the file has been partially updated, you must recover from a backup.

Data dictionary not updated
An error occurred during the create or reconfigure, so the data dictionary was restored to its previous state by recovering from the backup copy.

RECOVER DATA BASE FROM THE LAST BACKUP
An error occurred during the reconfigure after it started to write over the database file. Since the file is now unusable, you must recover it from a backup.

Data base is empty, use create
Reconfiguring an empty database is not allowed. Use the Create function instead.

Incorrect type of file
One of the block or character files specified in Volume Maintenance as a database volume is incorrect. They are

probably interchanged. This must be corrected before you can create or reconfigure the database.

Special file not found
One of the block or character files specified in Volume Maintenance was not found. These files must exist before you can create or reconfigure the database.

Can't open file - rddb
Either file.db or file.dbr cannot be opened. Make sure that the files are present and updatable. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

Can't open file.db - bufinit
file.db cannot be opened. Make sure that the file is present and updatable. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

Can't open file.db - savefree
file.db cannot be opened. Make sure that the file is present and updatable. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

Can't open file.db - movdown
file.db cannot be opened. Make sure that the file is present and updatable. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

Can't open reconf.db - getfree
This temporary file is missing. If you can duplicate the

problem, please forward a problem description and diskette containing file.db and unify.db to your nearest Radio Shack Computer Center.

Can't open reconf.db - savefree
This temporary file is missing. If you can duplicate the problem, please forward a problem description and diskette containing file.db and unify.db to your nearest Radio Shack Computer Center.

Can't open volume - readtemp
One of the database volumes cannot be opened for update. Make sure that all the device special files for your data base volumes are present and updatable.

Incorrect volume block
One of the block volume specifications from Volume Maintenance does not describe a block device. Make sure that all database volume block device specifications refer to an existing block device.

Incorrect character volume
One of the character volume specifications from Volume Maintenance does not describe a character device. Make sure that all database volume character device specifications refer to an existing character device.

Reconfiguration terminated
You have chosen not to continue the reconfiguration, so the process has been stopped.

Too many record types defined
More than 256 record types have been found in unify.db. Since Schema Maintenance prevents this from happening, unify.db is probably unusable and should be recovered from a backup.

Unable to create database file!
If you are creating a database file, make sure that you have write privilege in the database directory, sufficient free space, and available inodes. If you are creating a raw database file, make sure that the program RECONF is in a

directory owned by root, is owned by root, and has mode Ø4455 (executable, set user id on execution).

Unable to open data base volume
One of the database volumes cannot be opened for update. Make sure that all the device special files for your data base volumes are present and updatable.

No room for segment n of record type A
allocation error - getseg
Reconfigure Data Base has run out of space on one of the database volumes. This could happen as a result of increasing the lengths of records. Add another volume, or increase the length(s) of the current volume(s).

data block not found - recphys
A data block that is supposed to be present could not be found. file.db is probably unusable and should be recovered from a backup.

fields nested too deeply
Nested references to other COMB fields are generating fields more than five levels deep. Reduce the level of nested reference fields in the schema.

insufficient memory to compile schema!
The address space of the hardware you are using is not large enough to support the number of record types and fields in your schema. Either reduce the size of the schema, or use a computer with more than 64K data space.

invalid field number - procfld
An invalid field number has been found in unify.db. The file is probably unusable and should be recovered from a backup.

name generation stack overflow
The stack allocated for generating COMB reference field names has been overflowed. Reduce the length of the affected field names, or reduce the level of reference field nesting.

name stack overflow - entrnm
The stack allocated for storing record and field names and synonyms has been overflowed. Reduce the length of record and field names and synonyms.

read error - readrec
An unexpected end of file has been reached on file.db. The file is probably unusable and should be recovered from a backup.

reconf.tt botched - readtt
An unexpected end of file has been reached on this temporary file. If you can duplicate the problem, please forward a problem description and diskette containing file.db and unify.db to your nearest Radio Shack Computer Center.

reconf.tt missing - readtt
This temporary file could not be found. If you can duplicate the problem, please forward a problem description and diskette containing file.db and unify.db to your nearest Radio Shack Computer Center.

xxxx record type has too many fields
More than 256 fields have been found for the indicated record type in unify.db. Since Schema Maintenance prevents this from happening, unify.db is probably unusable and should be recovered from a backup.

referenced field does not exist
A referenced field could not be found in unify.db. The file is probably unusable and should be recovered from a backup.

relation table overflow
This is an internal error. If you can duplicate the problem, please forward a problem description and diskette containing file.db and unify.db to your nearest Radio Shack Computer Center.

translate table overflow - updtt
The sum of fields and relationships in some record type is more than 256. Reduce the number of fields or explicit relationships so that the sum is less than 257.

translate table read error - finishtt
There was an unexpected end of file on a temporary file. If you can duplicate the problem, please forward a problem description and diskette containing file.db and unify.db to your nearest Radio Shack Computer Center.

translate table write error - recterm
An error occurred in writing to a temporary file. There is probably no diskette space left in the database directory. Remove some files in the file system to free some space.

translate table write error - ttwrit
An error occurred in writing to a temporary file. There is probably no diskette space left in the database directory. Remove some files in the file system to free some space.

unable to create reconf.tt - initdbh
An error occurred in creating a temporary file. Make sure that you have write privilege in the database directory.

unable to open database file!
file.db cannot be opened. Make sure that it is present and updatable. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

unable to open file.db
file.db cannot be opened. Make sure that it is present and updatable. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

unable to open file.dbr - wrtdbhd
file.dbr cannot be opened. Make sure that it is present and updatable. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

unable to open reconf.db - wrtdbhd
This temporary file is missing. If you can duplicate the problem, please forward a problem description and diskette containing file.db and unify.db to your nearest Radio Shack Computer Center.

write error - writebuf
An error has occurred in writing the new format database file. If you are reconfiguring on diskette, you could be out of diskette space, or there could be a hard disk error. Delete some files to free some diskette space and try again. If you are reconfiguring to the backup device, you could have a faulty diskette, or the number of blocks per volume could be set incorrectly. Try using a new diskette in place of the faulty one. If the problem persists, use System Parameter Maintenance to make sure that the BLOCKS/VOLUME parameter is set correctly.

Secondary Index Maintenance Messages

CAN'T INDEX A COMBINATION FIELD
B-trees are not allowed currently on COMB type fields.

FIELD NAME NOT FOUND
The field name is not in the data dictionary. Make sure that you have spelled it correctly and are using the "short" field name.

THIS IS THE FIRST INDEX TO BE CREATED
For your information only.

WARNING: DBMS RECORD 'INDEX DROPPED', BUT IT WAS NOT FOUND
The field was listed as having an index, but the index file itself was not found. This is a warning only. You should probably check the integrity of your database file system using fsck(1).

INDEX ALREADY EXISTS
You are trying to index a field that is already indexed.

INDEX DOES NOT EXIST
You are trying to drop an index for a field that is not indexed.

Volume Maintenance Messages

Volume already exists
There is already a database volume with this name. Each name must be unique.

One line must contain an 'x' in the RT column
If you have specified database volumes, you must indicate which one is the root volume. The root volume is indicated by placing an 'x' in the RT column.

Can't add on a non-empty line
The add command is only valid on an empty (blank) line.

Can't delete an empty line
The delete command is only valid on a non-empty line.

Can't modify an empty line
The modify command is only valid on a non-empty line.

Enter [a]dd, [m]odify or [d]elete
The valid commands are add, modify, or delete.

Write Data Base Backup Messages

Can't open data base file - BUDB
The database file, file.db, could not be opened for reading. The following things could be wrong: you might be in the wrong directory; you might not be allowed to read file.db; DBPATH might be set incorrectly; or fil.db might not exist. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

Can't open data base volume
One of your raw database volumes is inaccessible. Check for the conditions listed in the previous error message, and also make sure that the special file pointers in /dev exist and are readable.

Can't open data dictionary
The data dictionary file, unify.db, could not be opened for reading. The following things could be wrong: you might be in the wrong directory; you might not be allowed to read unify.db; DBPATH might be set incorrectly; or unify.db might not exist. file.db and file.dbr should be linked. To check this, do an ls-|file.db file.dbr to see if both files have the same inode number. If they do not, remove file.dbr and link them again, using the command: ln file.db file.dbr.

Read error - BUDB
A hard read error occurred on the diskette. Try writing the backup again.

The backup device environment variable (BUDEV) is not set
This environment variable must contain the complete path name of the backup device, and it must be exported.

Write error - BUDB
A hard write error occurred on the backup device. Try writing the backup again, using a different diskette.

Read Data Base Backup Messages

Diskette read error - REDB
A hard read error occurred on the backup. Try reading it again. You might have to use a different diskette.

Can't create data dictionary
The file, unify.db, could not be created. Make sure that you have write privilege in the database directory and write privilege for the file. Then try again.

Can't open data base volume
The file, file.db, does not exist or is not writable. Change the mode (chmod(1)) or owner (chown(1)), and try again. If your database is on a raw device(s), make sure that the appropriate special files exist and are writable. Try performing a Create Data Base, and then read the backup again.

Error writing data dictionary
You are probably out of space in the database file system. Try removing some files, and then read the backup again. In rare cases, this could be a bad block on your diskette, in which case you must reformat the diskette.

The backup device environment variable (BUDEV) is not set
This environment variable must contain the complete path name of the backup device, and it must be exported.

Write error - REDB
You are probably out of space in the database file system. Try removing some files, and then read the backup again. In rare cases, this could be a bad block on your diskette, in which case you must reformat the diskette.

Screen Maintenance Messages

Field already exists
This screen field name is already in the data dictionary.

No more pages
This is the last page of screen fields.

No previous page
This is the first page of screen fields.

Screen already exists
This screen name is already in the data dictionary.

Screen unknown
This screen name does not exist in the data dictionary.

'X' coordinate must be between Ø and 79
You entered an invalid X coordinate.

'Y' coordinate must be between Ø and 23
You entered an invalid Y coordinate.

Field has invalid type
This is an internal error. The field type of the database field associated with this screen field is invalid. Try to delete the screen field and add it again.

Field has not been compiled
This is not a valid database field, because it has not yet been through a Create or Reconfigure Data Base.

Field unknown
This database field name is not in the data dictionary.

Invalid type
The entry in the TP column must be a valid database field type.

Length must be >= 1
The length in the LEN column must be greater than or equal to 1.

Please enter [M]od, [D]el, or [Q]uit
The valid commands are modify, delete, or quit.

Test Screen Messages

Screen unknown
This screen name is not in the data dictionary.

Process Screen Messages

Only 6Ø fields/screen allowed; cannot process
There is a limit of 6Ø fields per screen form. Delete some of the fields to use the form.

Screen unknown
This screen name is not in the data dictionary.

Screen Reports Messages

At least one response must be Y
To receive any output on the printer, at least one of the responses must be a Y.

Can't open printer
The printer cannot be accessed. Make sure that it is turned on, on-line, and loaded with paper.

Screen unknown
This screen name is not in the data dictionary.

Restore Screen Messages

Screen already exists
There is already a screen form with this name in the data dictionary, so you cannot recover it. Rename the .q file, and restore the screen form with that name.

Can't open xxxx
This screen form binary file cannot be found or is not readable.

Create Default Screen Form Messages

An ENTER Screen with the same name already exists
The default name of a form is the same as the record name, and there is already a form by this name registered with ENTER. If you want to create the default form with this program, delete the registration for the other form, and then rename it.

Screen already exists
There is already a form in the data dictionary with the same name as this record type. Rename the other form.

A field is not compiled, reconfigure or create the data base
Every field in the record type must have been through either Create or Reconfigure Data Base.

No fields in record
There are no fields in this record type, so it is impossible to create a screen form for it.

Enter Screen Registration Messages

ENTER screen already exists
There is already an ENTER screen with this name.

Screen unknown
There is no screen form with this name.

ENTER screen unknown
There is no ENTER screen registration record for this screen form name.

A target record was not entered. The screen is not registered
An ENTER screen must have a target record, which was not specified. The registration record has been deleted.

Record unknown
There is no record type in the database with this name.

Messages When Using ENTER Screens

Invalid key
The key of this record refers to another record type in the database, and the key you entered does not exist in the reference file.

Record already exists
This key is a duplicate of another record of the same type.

xxxx not found
This field is a reference field, and the value you entered does not exist in the related file.

Access privilege does not allow deletion
There is a field in this record type protected by Field Level Security, so you are not allowed to delete it.

Record is in use by another process
The current record is locked by some other program, so you cannot update it until it is released.

Record not deleted due to related records
This record has other records related to it. Delete all the related records before deleting this one.

No matches
There are no records in the file that match the selection criteria you entered.

No more records
You are at the last record in the current set, or you have deleted the last record in the current set.

No previous record
You are at the first record in the current set.

This record has been deleted
Someone at another terminal has deleted the current record.

This record is currently in use on another terminal
You cannot modify or delete the current record because someone at another terminal is modifying it.

Error in entmain (n), empty record added
This is an internal error. ENTER tried to delete the current record but was unsuccessful. Try to find the empty record and delete it.

Cannot change key due to references
Other records reference the key of the current record, so it cannot be deleted. Add a new record with the desired key, change the referencing record to refer to this new record, and then delete the desired record.

Duplicate keys not allowed
You tried to change the key to the same value as another record in the file. All primary keys must be unique.

Field is write protected
The current field is protected by Field Level Security, so you cannot update it using ENTER. You can customize ENTER to unlock the field, write a program that unlocks and updates the field, or unprotect the field.

Deadlock detected from system locking
Two processes are competing for the same records and are in a deadlock situation. Return to the menu handler and begin the current operation again.

Undefined return from pfield
This is an internal error. An undocumented return from pfield has been detected. Reaccess the record, and make sure that it looks right.

Illegal entry
You entered an invalid command. Choose from the set of options displayed.

Record too large
The target record is larger than 2Ø48 bytes, the largest record that ENTER can handle.

Target record key is missing or incomplete
The screen form does not contain the complete key of the target record. Use Screen Maintenance to correct the screen form design.

C Language Interface Messages

Illegal value
An AMOUNT field contains characters other than digits, a decimal point or a minus sign.

Value is too large
The value entered is too large for the schema definition of the AMOUNT or FLOAT field.

Illegal date
The value entered for a DATE field has an invalid format or is not a valid date. The correct format is mm/dd/yy, where mm is a month between 1 and 12; dd is a day between 1 and 28, 29, 3Ø, or 31, depending upon the month and year; and yy is the last two digits of the year from ØØ to 99.

Illegal integer value
The value entered for a NUMERIC 1-4 field contains non-numeric characters.

Illegal numeric value
The value entered for a NUMERIC 5-9 field contains non-numeric characters.

Illegal time
The value entered for a TIME field has an incorrect format or is not a valid time. The correct format is hh:mm, where hh is a number between Ø and 23 and mm is a number between Ø and 59.

Record key is in use
The key of this record cannot be changed because there are other records that reference it.

Duplicate key not allowed
The key of this record cannot be changed to this value because there is another record in the database with the same key.

Illegal entry
You entered an invalid command. Choose from the set of options displayed.

System is locked: Program exit
A program has prematurely stopped in the middle of a database update and left a lock file in the database directory. Refer to the documentation for lock in section 1Ø of this manual for further information.

APPENDIX D
MENU MAP

This Menu Map lists the different menus used by Unify. Each option on the menu is included, plus its program name and an explanation of its function.

System Menu -- [sysmenu]

1. Schema Maintenance [schent]
   Use to create a database. Includes options to add, modify, delete, or inquire about the records and their fields.

2. Schema Listing [shlst]
   Use to print a listing of all the records and fields.

3. Create Database [crdb]
   Use to create a clean and empty database.

4. SFORM Menu [sfmenu]
   Use to display a menu of different options for creating or using screens.

5. Enter Screen Registration [entmnt]
   Use to register each screen after its creation.

6. SQL - Query/DML Language [sql]
   Structured Query Language. Uses clauses and keys to retrieve information from the database.

7. Listing Processor [lst]
   Selection and formatting language. Use it to produce file listings and printouts. Uses two types of prompts: (*) for selecting records and (-) for formatting records.

8. Database Test Driver [sys92Ø]
   Use only when operating with the TRS-XENIX Development System. It uses the record and explicit relationship

functions in an interactive mode after a database file is created.

9. MENUH Screen Menu [scrmen]
   Displays a menu which contains programs that control database security, help documentation, and language interfaces with the database.

1Ø. MENUH Report Menu [rptmen]
   Prints a list of all menus in the data dictionary. The menus are listed in alphabetical order by name, and each menu is followed by the selection options it contains.

11. Reconfigure Database [scom]
   After each change (addition, deletion, modification) to the database, execute this program. It reformats the database file and updates the data dictionary. No one should use the Unify application while the database is being reconfigured.

12. Write Database Backup [budb]
   Use to make a backup of the UNITRIEVE database file and the data dictionary in the current directory.

13. Read Database Backup [redb]
   Use to read a backup diskette (written by the Write Database Program) to restore UNITRIEVE database and data dictionary files.

14. Database Maintenance Menu [dbmenu]
   Displays a menu of different options for menu maintenance.

MENUH Screen Menu -- [scrmen]

1. Executable Maintenance [execmnt]
   Use to add, modify, and delete executable files.

2. Menu Maintenance [menumnt]
   Use to inquire, add, modify, and delete menus and menu lines. The screen contains an entry area for menus and a multi-line entry area for menu lines.

3. Group Maintenance [grpmnt]
   Use to inquire, add, modify, and delete employee groups and access privileges. The screen contains an entry area for menus and a multi-line entry area for menu lines.

4. Employee Maintenance [empmnt]
   Use to inquire, add, modify, and delete employees and access privileges. Also use it to indicate differences between one employee's privileges and those of the employee's group.

5. Enter Help Documentation [enthdoc]
   Use to create a text file associated with a particular program, ENTER screen, or menu.

6. Program Loading [lfilegen]
   Use only when operating with the TRS-XENIX Development System. Use it to control loading of executable modules for an application system by registering each program with MENUH in Executable Maintenance.

7. System Parameter Maintenance [parmnt]
   Stores basic parameters for the menu system, for example, information about the super user, the system title, and mnemonic codes.

MENUH Report Menu -- [rptmen]

1. Executable Listing [lexec]
   Prints a report of all executables in the data dictionary including the programs.

2. Menu Listing [lmenu]
   Prints an alphabetical list of all menus in the data dictionary, followed by the selection options each menu contains.

3. Group Listing [lgrp]
   Prints a list of all employee groups in the data dictionary in alphabetical order by group id, followed by a list of the programs to which the group has access.

4. Employee Listing [lemp]
   Prints a list of all employees in the data dictionary in alphabetical order by employee login id, followed by a list of the programs in which the employee has different privileges from the group.

5. Print Help Documentation [prthdoc]
   Prints the Help documentation associated with a menu or program. You need to enter the name of the program.

Database Maintenance Menu -- [dbmenu]

1. Field Security Maintenance [fldsec]
   Allows you to define separate read and write passwords for each database field.

2. Process Field Passwords [procpass]
   Processes the field-level password specifications to produce the binary password file.

3. Index Maintenance [idxmnt]
   Use to add a secondary (B-tree) index or to drop an existing index. No one should update the field(s) when this program is in use.

4. Volume Maintenance [volmnt]
   Use to describe the extents (uslames) for the database. A database can be stored in one or more contiguous extents of diskette space.

5. Hash Table Maintenance [rekey]
   Rebuilds the hash table from scratch. Use when a record type cannot be accessed by the primary key but can be found by scanning the file or by using a secondary key.

6. Explicit Relationship Maintenance [repoint]
   Rebuilds the explicit relationships in the database. Use when the explicit relationship functions (unisort, makeset, setsize, nextrec, prevrec) quit functioning or start returning inconsistent results.

7. Database Statistics [dbstats]
   Prints statistics about various database parameters.

8. Hash Table Statistics [hts]
   Prints statistics about various hash table parameters.

SFORM Menu -- [sfmenu]

1. Screen Entry [sfmaint]
   Use to inquire, add, modify, and delete screen forms and screen fields.

2. Test Screen [sfsamp]
   Use to test a screen without writing a program to display it. Also checks the position of the prompts and data entry points.

3. Process Screen [sfproc]
   Creates the screen which was entered during screen entry. Execute this option after adding a new screen or modifying an existing one.

4. Screen Reports [sfrep]
   Prints a copy of the screen, a list of all the screen field information or both.

5. Restore Screen [sfrestr]
   Recovers screen forms which are accidently deleted from the data dictionary or re-orders the prompts on a screen.

6. List Screens [sfslist]
   Displays the names of all screen forms in the data dictionary.

7. Create Default Screen Form [cdsf]
   Use to create, process, and register a screen form with ENTER. By using this program, there is no need to complete Process Screen and Screen Registration.

INDEX

access privileges 4Ø, 47, 6Ø-61, 65-66, 68-7Ø, 84-85, 12Ø, 123, 2Ø2, 513-514, 532
aggregate functions 226-228, 271-272
arithmetic expressions 222-224, 226, 234-235, 272, 274, 278, 29Ø, 3ØØ-3Ø1
avg 226, 271-272, 278-279

backup
  write database 79, 1Ø3, 11Ø, 128-133, 526
  read database 79, 1Ø3, 131-133, 527
boolean expressions 216-217, 273, 276-277, 28Ø, 29Ø-291
B-trees 27, 111-113, 36Ø, 382, 4Ø1-4Ø7, 525

cartesian product 238, 24Ø
clause
  order by 224-226, 281
  group by 227, 228-231, 276-277
  having 234-238, 277-281
  insert 251
  into 252-253, 281
  select 21Ø-212, 27Ø-272
  from 21Ø-212, 272
  where 212-213, 235-236, 241-242, 273-276
column headings 3Ø2, 3Ø6-31Ø
combined fields 94, 96-97, 1Ø1, 159, 164, 168, 222, 474, 517-518, 522
compiling programs 32Ø-323
concatenation operator 29Ø, 3Ø1
control keys 35-36, 168-17Ø
count 226, 271-272, 278-279
create database 1Ø3-1Ø6, 114-115, 519
CTRL E 35-36, 17Ø
CTRL P 35
CTRL X 35-36, 169-17Ø
CTRL Z 35-36, 17Ø
current record 319

data dictionary 2Ø-21, 23-26, 32, 37, 5Ø, 6Ø, 74, 81-82, 87-88, 1Ø3, 1Ø6, 138, 148, 155, 157, 325, 457, 511-512, 513-515, 527-528
database test driver 313-315
data types 96
debugging programs 324-325
default
  field values 174, 182, 188
  screen form 158-161
development system 17-2Ø
dimensions, screen 149
directory
  application 27,29

structure 2Ø-25, 27-3Ø, 5Ø, 56, 72, 76, 87
system 27-28
tutorial 28
disk configuration 16-17
diskette 15, 19-2Ø, 11Ø, 13Ø-136

editing
fields 174, 18Ø, 183, 188
queries 248-249, 282
editor commands, vi 33, 5Ø1-51Ø
employee maintenance 65-71
entry point, system 61-62, 66, 68, 79, 516
environment variables 13, 3Ø-32, 5Ø-51
error messages 511-535
errors
fatal 324-337
SQL 255-268
trapping fatal 326
executable maintenance 47, 52-56, 75
executables 43-49, 7Ø-77, 81-82, 2ØØ-2Ø4, 323, 512
expected number of records 91, 99, 1Ø9, 125
explicit relationships 95, 136-137, 163, 174, 313, 318, 325, 36Ø, 382, 4Ø7, 523
expressions
arithmetic 222-224, 226, 234-235, 272, 274, 278 29Ø, 3ØØ-3Ø1
logical 213-217, 273, 275-277, 28Ø-281, 29Ø-291

fields
combined 94, 96-97, 1Ø1, 159, 164, 168, 222, 474, 517-518, 522
editing 174, 18Ø, 183, 188
inexact matching on non-string 171
listing 3Ø2-3Ø5
file
selection 287, 289-29Ø, 298-299, 36Ø, 372, 375-38Ø
specification 138, 139
file.db, file.dbr 26, 28-29, 3Ø, 1Ø4, 115, 123, 324, 457 52Ø, 524
file.h 26, 2Ø2, 317, 321, 475, 487
files
raw 26, 1Ø4, 113
saving data to XENIX files 25Ø-251
screen .h 27, 144, 151
screen .q 27, 144, 151, 155
Unify 13, 25-27
formats
internal data 318
print 3Ø2-3Ø5, 434
forms
ENTER screen 158-159
query by 169-173
screen 143
function
input 174-175, 182, 189-19Ø
post-processing 174-175, 183, 188, 192-193

pre-processing 174-175, 181, 188, 195
screen initialization 174-175, 179, 189
screen termination 174-175, 180

hash table 95, 107, 109-111, 117, 125-127, 136, 325, 407
heading
system 78-79
ENTER 166-167
headings
column 302, 306-310
menu 58-59
program 55-56
report 309
help documentation 16, 28-30, 32, 42, 71-72, 85, 511

id, login 22, 40, 50-51, 67, 511
inclusion, set 218-220, 232
indexes, secondary 27, 111-113, 401, 525
inexact matching on non-string fields 171
input function 174-175, 182, 189-190
insert clause 251
installation 14-20, 32
internal data formats 318

join
natural 287, 290-291
self 243-245
join queries 238-243

key, primary 88, 95, 107, 111, 164, 168, 170, 220
keys
control 35-36, 168-170
secondary 111
keywords 205-206, 254

listing
fields 302
processor 287-311
loading
database 137-141, 251-253
ENTER 197-200
program 45, 72-77, 320-323
LOCKl 74
logical expressions 213-217, 273, 275-277, 280-281, 290-291
login id 22, 40, 50-51, 67, 511

maintenance
database 103-118
employee 47, 65-70
executable 47, 52-56
field 92-97
group 47, 60-65
menu 50, 57-60, 156, 159
parameter 47, 78-80
record 89-92
screen form 145-149, 528
volume 113-118, 526
matching partial string 213
max 226, 271-272, 278-279

menu
  headings 58-59
  maintenance 5Ø, 57-6Ø, 156, 159
  prompts 6Ø
menu handler parameters 46-49
menucall 48
menus, using 38-39
min 226, 271-272, 278-279
multiple databases 29-3Ø, 32

natural join 287, 29Ø-291
nested queries 23Ø-234, 236, 255
null field values 318

operator
  between 217, 242
  concatenation 29Ø, 3Ø1
  not 217-218
  range 172
  substring 29Ø, 3Ø1
  unique 22Ø-222, 224, 271
operator precedence 216, 218, 29Ø, 3Ø1

parameter maintenance 47, 78-8Ø
parameters 3Ø-32
partial string matching 213
password 119-123
permissions 24, 29-3Ø
post-processing function 174-175, 183, 188, 192-193
pre-processing function 174-175, 181, 188, 195
primary key 88, 95, 1Ø7, 111, 164, 168, 17Ø, 22Ø
print formats 3Ø2-3Ø5, 434
printing reports 81-85, 153-154
privileges, access 4Ø, 47, 6Ø-61, 65-66, 68-7Ø, 84-85, 12Ø 123, 2Ø2, 513-514, 532
process screen form 11Ø, 151-152, 158-161
processor
  listing 287-311
  selection 287-291, 297
programs
  compiling 32Ø-323
  debugging 324-325
prompts
  menu 6Ø
  screen form 143, 149, 413

queries
  editing 248-249, 282
  join 238-243
  stored 249-25Ø, 283
  variable 245-248
query by form 169-173
query scripts 249-25Ø, 269, 289

range operator 172
raw files 26, 1Ø4, 113
read database backup 79, 1Ø3, 131-133, 527
reconfiguration 79, 88, 1Ø2, 1Ø6-111, 115, 136, 325, 457, 519

record
current 319
locking 168, 350-351, 358-359
maintenance 89-92
target 163, 166, 169, 202, 511, 531
records, expected number of 91, 99, 109, 125
recovery, database 128-136, 519
reference field 95, 164, 291, 301, 313, 523
relationships, explicit 95, 136-137, 163, 174, 313, 318, 325, 360, 382, 407, 523
report
headings 309
scripts 165, 167, 287, 299
restore screen 155-156
root volume 115, 117
runtime system 17-20

saving data to XENIX files 250-251
schema 88-89, 97-102, 103, 124-125, 136, 202, 291, 317, 318, 338, 398-399, 474, 487, 522
screen
restore 155-156
test 150
screen dimensions 149
screen field name 143, 147-148
screen form
default 158-161
maintenance 145-149, 528
process 110, 151-152, 158-161
prompts 143, 149, 413
screen forms, ENTER 143, 148, 156, 158-159, 163
screen .h files 27, 144, 151
screen initialization function 174-175, 179, 189
screen .q files 27, 144, 151, 155
screen termination function 174-175, 180
screens, using ENTER 168-173
scripts
query 249-250, 269, 289
report 165, 167, 287, 299
secondary
indexes 27, 111-113, 401, 525
keys 111
security
field 119-123, 260, 269, 283, 289, 297, 311, 340
program 21, 40, 60-70, 123
select clause 210-212, 270-272
selection
file 360
processor 287-291, 297
self join 243-245
separator, field 251, 284

shell, TRS-XENIX 29, 31, 36, 39, 5Ø-51, 54, 134, 137, 248-249, 25Ø-251, 285, 289, 299, 486
sorting 224-225, 276, 281, 299, 3Ø6
specification file 138, 139
SQL
  errors 255-268
  help 2Ø7-2Ø9, 286
statistics 124-127
stored queries 249-25Ø, 283
subset 287, 289, 292-296, 36Ø
substring operator 29Ø, 3Ø1
sum 226, 271-272, 278-279
super-user 21-23, 25, 37, 4Ø-41, 47, 74, 78-8Ø, 458, 515
sysrecev 43-44, 46-48, 54, 77, 2Ø3
system
  development 17-2Ø
  runtime 17-2Ø

target record 163, 166, 169, 2Ø2, 511, 531
termcap 13, 32-36, 17Ø, 413
test screen 15Ø
totaling 299, 3Ø6-3Ø7
trapping fatal errors 326
tutorial directory 28

ucreate 24, 3Ø
umove 22-23
unify files 13, 25-27
unify.db 21, 23, 25-26, 28-29, 32, 123, 457, 521
unique operator 22Ø-222, 224, 271
usave 133-136
using
  ENTER screens 168-173
  menus 38-39

values
  default field 174, 182, 188
  null field 318
variable queries 245-248
vi editor commands 33, 5Ø1-51Ø
volume maintenance 113-118, 526

write database backup 79, 1Ø3, 11Ø, 128-133, 526

XENIX files, saving data to 25Ø-251