Rocket UniVerse

System Description

Version 11.3.5

January 2023
UNV-1135-SYS-1

Notices

Edition

Publication date: January 2023
Book number: UNV-1135-SYS-1
Product version: Version 11.3.5

Copyright

© Rocket Software, Inc. or its affiliates 1985-2023. All Rights Reserved.

Trademarks

Rocket is a registered trademark of Rocket Software, Inc. For a list of Rocket registered trademarks go to: www.rocketsoftware.com/about/legal. All other products or services mentioned in this document may be covered by the trademarks, service marks, or product names of their respective owners.

Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

License agreement

This software and the associated documentation are proprietary and confidential to Rocket Software, Inc. or its affiliates, are furnished under license, and may be used and copied only in accordance with the terms of such license.

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country	Toll-free telephone number
United States	1-855-577-4323
Australia	1-800-823-405
Belgium	0800-266-65
Canada	1-855-577-4323
China	400-120-9242
France	08-05-08-05-62
Germany	0800-180-0882
Italy	800-878-295
Japan	0800-170-5464
Netherlands	0-800-022-2961
New Zealand	0800-003210
South Africa	0-800-980-818
United Kingdom	0800-520-0439

Contacting Technical Support

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to www.rocketsoftware.com/support.

In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Contents

Notices 2

Corporate information 3

Chapter 1: The UniVerse environment 10
- Main features 10
  - UniVerse system structure 10
  - Main components 11
  - UniVerse command line help 11
  - The UniVerse command processor 11
  - Other UniVerse language processors 12
  - UniVerse BASIC 12
  - RetrieVe 12
  - UniVerse SQL 13
  - Other UniVerse utilities and processes 13
- The operating system and UniVerse 13
  - UniVerse terminal I/O and terminfo 13
- UniVerse files 14
  - UniVerse file structure 14
    - Nonhashed files 14
    - Hashed files 14
    - Dynamic files 15
    - B-Tree files 15
- The XDEMO account 15
  - XDEMO files 15
- VOC file 16
- UniVerse commands 16
- Review of terms 17

Chapter 2: UniVerse accounts 19
- Your UniVerse account 19
  - Windows UniVerse accounts 19
    - User names and domain names 19
    - Local logon policy 20
  - UniVerse account flavors 20
  - UniVerse login entry 20
  - ON.EXIT entry 21
  - ON.ABORT entry 21
  - Creating a UniVerse account 21
  - Updating your account 21
- UNIX and UniVerse account differences 22
  - Different environments for different users 22
  - Your UNIX account 22
    - The standard UNIX shell environment 22
    - Moving from one UNIX shell to another 23
    - Entering UniVerse from a UNIX shell 23
    - Shell environment initialization 23

Chapter 3: The command processor and the VOC file 24
- The command processor 24
  - Special character interpretation 24
    - Using a period and a hyphen in verbs and keywords 24
    - Displaying other special function characters 24
  - Using the interrupt, quit, and suspend keys 25

Phantom processes... 25
PHANTOM use that will consume a database license... 26
The VOC file... 26
Listing the VOC file contents... 27
VOC file record format... 27
Field 1 (F1)... 28
Fields 2 through 5 (F2, F3, F4, F5)... 28
Additional fields... 29
VOC entry types... 29
V: Verb... 30
R: Remote command entry... 31
Security subroutines... 32
Security subroutine example... 32
S: Stored sentence... 33
PA: Paragraph... 34
F: File definition... 35
VOC file pointers... 35
Creating synonyms... 36
Creating file pointers... 36
Q: Q-pointer... 36
K: Keyword... 37
Creating keyword synonyms... 37
PH: Phrase... 37
M: Menu... 38
PQ: ProVerb (PROC)... 38
X: User record... 39
RELLEVEL and STACKWRITE records... 39
INTR.KEY, QUIT.KEY, and SUSP.KEY records... 39
Tailoring the VOC file... 39
Pointing to records in the VOCLIB file... 40
How the VOC file is used... 40
Listing selected VOC file entries... 40
UniVerse sentence stack... 41
Saving the sentence stack... 41
Loading a saved or empty stack... 42
Using the GET.STACK command... 42
Using the SAVE.STACK command... 42
Using paragraphs... 42
Creating a paragraph... 43
Using inline prompts in paragraphs... 43
Controlling paragraph execution... 44
The IF command... 44
The GO command... 44
The LOOP command... 45
The DATA command... 45
Chapter 4: Creating a database... 46
Database structure... 46
Defining UniVerse files... 46
Creating the data file and Its dictionary... 47
Creating the database... 47
Designing the files... 48
File design objectives... 48
Guidelines for choosing file type 1 or 19 (nonhashed)... 48
Guidelines for choosing other file types... 49
Allocating space for the files... 49
Effect of modulo and separation on UniVerse performance... 49

Creating files... 49
CREATE.FILE prompts for parameters... 50
Modifying a VOC file definition... 51
Naming files... 51
Spaces and control characters in filenames... 51
Long file names... 52
Long record IDs... 52
Long record IDs in a type 19 file... 53
File types... 53
Nonhashed file (type 1 and type 19)... 53
Static hashed file (types 2–18)... 53
B-Tree file (type 25)... 53
Dynamic file (type 30)... 54
Distributed file... 54
Modulo and separation... 54
Modulo... 54
Separation... 55
Important effects of separation on a file... 55
Summary of file sizing... 55
Building the file dictionary... 56
Adding data to the data file... 56
Secondary indexes... 57
Creating and building secondary indexes... 57
Creating secondary indexes... 58
Building secondary indexes... 59
Deleting secondary indexes... 59
Enabling and disabling secondary indexes... 59
Listing secondary indexes... 60
Using secondary indexes with distributed files... 60
Accessing indexes on part files... 60
Accessing indexes using distributed files... 60
Examples illustrating corresponding dictionary entries... 61
Using secondary indexes from UniVerse BASIC... 62
Chapter 5: UniVerse file dictionaries... 63
UniVerse dictionary entries... 63
DICT.DICT file... 63
Kinds of dictionary entry... 63
@ID record... 64
Data descriptor... 64
I-descriptor... 64
Phrase... 64
Special phrases... 65
X-descriptor... 66
Structure of dictionary entries... 66
Dictionary field descriptions... 69
@ID field... 69
Field 1... 69
Field 2... 70
D- and I-descriptors... 70
Field 3 (D and I)... 70
Field 4 (D and I)... 70
Field 5 (D and I)... 71
Field 6 (D and I)... 73
Field 7 (D and I)... 73
Field 8 (D and I)... 74
A- and S-descriptors... 74

Field 3 (A and S)..... 75
Field 4 (A and S)..... 75
Field 5 (A and S)..... 75
Field 6 (A and S)..... 76
Field 7 (A and S)..... 76
Field 8 (A and S)..... 76
Field 9 (A and S)..... 77
Field 10 (A and S).....77
Using I-descriptors.....77
Defining I-descriptors..... 77
I-type expressions.....77
Field names in I-type expressions..... 78
@variables in I-type expressions.....79
@NI, the item counter.....79
@NB, the BREAK level number.....80
Numeric constants and literal strings in I-type expressions.....81
BASIC functions in I-type expressions.....81
Operators in I-type expressions.....82
Conditional expressions in I-type expressions..... 83
Compound I-type expressions.....83
Getting data from other files using the TRANS function.....84
Calling BASIC subroutines from I-type expressions..... 85
Using the TOTAL function and the CALC keyword..... 88
Generating histograms.....90
Compiling I-descriptors.....91
Chapter 6: UniVerse file structures.....92
Static hashed files..... 92
Dynamic files.....92
Changing dynamic file parameters..... 92
Important considerations..... 93
Distributed files.....93
The partitioning algorithm.....94
Creating and modifying distributed files..... 95
Creating a new distributed file.....95
Specifying the partitioning algorithm.....95
Adding a part file.....97
Adding a part file that belongs to another distributed file.....98
Changing the part number..... 98
Changing the partitioning algorithm..... 99
Removing part files..... 99
How UniVerse deletes a distributed file.....99
Listing part file information.....100
Examples..... 100
Verifying part files.....100
Example.....100
Rebuilding distributed files.....101
Example.....101
Type 1 and type 19 files.....102
Secondary indexes.....102
Multiple data files..... 103
Clearing multiple data files.....105
Changing multiple data filenames..... 105
Deleting multiple data files.....105
Long name support for multiple data files.....106
Pointers to files in other accounts..... 106
Creating file pointers.....107

Creating Q-pointers ... 107
Sizing dictionaries ... 107
UniVerse file protection ... 108
File locks and record locks ... 108
File locks ... 108
Record locks ... 108
Releasing locks ... 109
Implicit locks ... 109
Read locks ... 109
Write locks ... 109
Informational locks ... 109
Operating system file protection ... 109
Displaying permission values ... 110
Improving file performance ... 111
Verifying file structure ... 111
Controlling hashing ... 111
File maintenance commands ... 113
Analyzing dynamic files ... 114
Listing statistics for static hashed files ... 114
Analyzing any file type ... 115
File usage statistics ... 116
Showing how records are distributed in groups ... 117
Determining the best file structure for hashed files ... 118
Testing a new file structure ... 119
Resizing files ... 119
Changing file type ... 120
Monitoring the VOC file ... 120
Repairing damaged files ... 120
Cleaning up an account ... 121

Chapter 7: File triggers ... 122
Applying business rules ... 122
Using triggers ... 122
When does a trigger fire? ... 123
What events fire a trigger? ... 123
Creating triggers ... 123
UniVerse BASIC WRITE on existing record ... 123
UniVerse BASIC WRITE of a new record ... 123
UniVerse BASIC DELETE of existing record ... 124
Modifying triggers ... 124
Listing information about triggers ... 124
Trigger programs ... 125
Opening files ... 125
index-based subroutines ... 126
Programming example ... 127

Chapter 8: Peripheral devices ... 128
Redirecting output ... 128
Assigning devices ... 128
Unassigning devices ... 129
Setting printer parameters ... 129
Putting print jobs on hold ... 130
Printing RetrieVe output ... 130
Using the default printer ... 130
Changing print parameters before printing ... 130
Sending output to an assigned printer ... 131
Sending output to a hold file ... 131

Printing complete records ... 131
Printing to and from tape ... 131
Printing UniVerse BASIC output ... 132
The UniVerse spooler (UNIX) ... 132
Using the UniVerse spooler ... 132
Checking print job status ... 133
Changing print job status ... 133
Modifying print job characteristics ... 134
Using tape drives ... 134
Copying records from disk to tape ... 134
Copying records from tape to disk ... 134
Terminals ... 135
Displaying your terminal setting ... 135
Setting terminal characteristics ... 136

Chapter 9: Using ReVise ... 137
What is ReVise? ... 137
Specifying ReVise prompts ... 138
Entering data on different screens ... 139
Responses to ReVise prompts ... 140
Responding to the record ID prompt ... 140
Responding to singlevalued field prompts ... 140
Responding to multivalued field prompts ... 141
Responding to the change which line item= Prompt ... 142
Responding to the CHANGE= Prompt ... 142
Changing file dictionaries ... 143
Changing data files ... 145
Singlevalued fields ... 145
Unassociated multivalued fields ... 146
Associated multivalued fields ... 147
ReVise with an active select list ... 147

Chapter 10: Menus ... 149
Menus in UniVerse ... 149
File dictionary for menu files ... 149
Menu Maintenance menu ... 150
Creating the VOC Menu entry ... 151
Invoking menus ... 152
Creating formatted menus ... 152
Changing the default prompt, exit, and stop options ... 153
Nesting menus ... 153
Prompting within menus ... 154
Displaying menus in Motif format ... 154
Responding at any level of a Motif menu system ... 155
Choosing an option ... 155
Moving around the menus ... 155
Examples ... 155

Chapter 1: The UniVerse environment

UniVerse is a database management, development, and execution environment that is well suited for business applications. In a client/server environment, UniVerse functions as a relational database management system (RDBMS) server. It is simple enough to be used by people who are not programmers, yet powerful enough for experienced programmers to develop complex applications. As a tool for moving existing applications into open systems or for creating new applications, UniVerse has proven to be a productive development environment and an efficient execution environment. UniVerse provides easy-to-use access to complex open system technologies such as distributed processing, modern user interfaces, and the latest advances in hardware and operating systems.

Programmers can use the UniVerse BASIC programming language and the powerful facilities for stored command sequences and menu generation to develop business applications such as payroll, accounts receivable, inventory, sales tracking, and general ledger.

Users can use UniVerse SQL to define, query, and control the data in the database without changing existing UniVerse applications.

Main features

Some of the more powerful features of UniVerse are:

- Variable file sizes, record lengths, field sizes, and number of fields
- Unlimited number of files
- Several file access methods for the most efficient data storage, access, and retrieval
- Relational database facilities that let you create associations between fields in the same or in different files
- A dictionary-driven, interactive data entry processor for editing data in files
- Online help for any command in most of the command languages
- Facilities to create your own online help documentation
- English-like database retrieval language
- SQL database retrieval language
- Facilities to save command lines for future use and to create stored command sequences
- A powerful programming language with built-in database management extensions
- Facilities to create user-friendly menus to control your applications

UniVerse system structure

UniVerse is a self-contained environment that runs under the UNIX and Windows operating systems.

The UniVerse command processor interprets command lines, performs certain substitutions on command lines, and passes control to the proper process or utility. Other UniVerse processors, such as the UniVerse Editor, ProVerb, and ReVise, offer sets of commands tailored to their specific functions.

UniVerse also supports a procedural language that allows you to write a program, compile it, and then execute it. In addition to its own functions, UniVerse provides easy access to the operating system.

Main components

UniVerse comprises the following:

- Four interactive language processors: the UniVerse command processor, ProVerb, the UniVerse Editor, and ReVise
- A procedural language processor, UniVerse BASIC
- The BASIC run machine, which executes compiled BASIC programs
- A set of utilities and processes that the language processors and the run machine call on to perform their tasks
- RetrieVe, a database query processor and report writer
- The UniVerse file handler

UniVerse command line help

Use the UniVerse `HELP` command to get help about any UniVerse command, keyword, UniVerse BASIC statement, and so forth.

For information about the `HELP` command, enter `HELP HELP` at the UniVerse prompt. Enter `HELP` to display a list of entries, which are topics and commands about which you can get `HELP` information:> `HELP`

Use the Up and Down Arrow keys to navigate through this list. Press Enter to select an item and display explanatory text. Press Esc to exit the list.

Use the following syntax to display help about a specific command:

`HELP` *`command`*

When you use the `HELP` command, explanatory text appears with menu choices at the bottom of the screen. There are three choices:

Choice	Action
More	Shows the next page of `HELP` text.
List Commands	Displays a list of entries.
End Help	Exits `HELP`.

Inside the `HELP` display screen, use the Left and Right Arrow keys to select a choice. Press Enter to choose your selection.

The UniVerse command processor

The UniVerse command processor accepts commands from the terminal or other sources and either processes the command itself or calls another UniVerse or system process. When you first enter the UniVerse environment, the command processor is in control of the terminal.

Other UniVerse language processors

ProVerb

ProVerb is a UniVerse processor that interprets stored command statements. It is a good tool to use for minor programming tasks that use the resources of UniVerse. For example, you can use ProVerb to execute a proc that prints a daily sales report.

Editor

The UniVerse Editor is a line-oriented editor that you can use to add, change, and delete records in files.

ReVise

ReVise is a dictionary-driven data entry processor that you can use to add, change, and delete data in files. ReVise uses information stored in file dictionaries to provide prompts that are specific to the data file. This information can also be used to convert input format and validate input.

UniVerse BASIC

The UniVerse BASIC procedural language is an enhanced version of Pick BASIC. UniVerse BASIC source programs are created using the UniVerse Editor or any other suitable editor, and are stored as records in a UniVerse file. The `BASIC` command is used to compile the programs.

The BASIC compiler generates a new record in another file containing object code, which the UniVerse run machine can execute. Programs can also be cataloged in a local or system catalog, allowing them to be used as commands or as subroutines by other BASIC programs. For more information, see *UniVerse BASIC*.

RetrieVe

RetrieVe is one of the most powerful and versatile processors of the UniVerse database management system. It is used to process data in the database and to generate reports. All the command and language processors have facilities to call the RetrieVe processor. RetrieVe includes the following features:

- Record selection for processing by other commands
- Report formatting
- Sorting
- Data field manipulation capabilities

You can use command processor, Editor, and ReVise commands, as well as UniVerse BASIC statements and functions, to access RetrieVe functions. RetrieVe contains the routines called by the command processor, Editor, ReVise, and the run machine to format data according to dictionary definitions and to select records from a data file that meet certain criteria.

The commands that call on RetrieVe to list, sort, select, count, and in other ways process UniVerse files are called RetrieVe commands. For more information about RetrieVe, see the *Guide to RetrieVe*.

UniVerse SQL

UniVerse SQL is an enhanced version of today's industry-standard database language, seamlessly integrated into the UniVerse environment. It is both a database language and a set of capabilities. Using UniVerse SQL you can query and update data in UniVerse files as well as in SQL tables. You can also use UniVerse SQL in client programs as well as interactively.

UniVerse SQL provides the following enhancements (among others) to the UniVerse environment:

- Subqueries that let you nest queries
- Relational joins that let you work with data from more than one file or table in a single command
- Added database security and integrity

UniVerse SQL conforms to the ANSI/ISO 1989 standard established for SQL, enhanced to take advantage of the extended relational database structure of UniVerse. In contrast to first-normal-form (1NF) databases, which can have only one value for each row and column position (or cell), UniVerse is a nonfirst-normal-form (NF2) database, which can hold more than one value in a cell. UniVerse also supports nested tables called associations, which are made up of a group of related multivalued columns in a table. For complete information about UniVerse SQL, see the *UniVerse SQL Reference*, the *UniVerse SQL User Guide*, and *UniVerse SQL Administration for DBAs*.

Other UniVerse utilities and processes

The language processors and the run machine call UniVerse utilities and processes to create and delete files, display the system status, locate records in UniVerse files, control concurrent access to records or files, check file size, and so on. UniVerse also calls on system processes and utilities for certain tasks.

The operating system and UniVerse

UniVerse is a group of programs that runs in the operating system environment. However, because the operating system environment can be invisible to the end user, UniVerse can be perceived as the operating environment.

UniVerse has its own command processor, with a command vocabulary that includes many operating system commands and many data management commands that cannot be accessed from the operating system. UniVerse also has its own login procedure and account structure.

UniVerse terminal I/O and terminfo

On UNIX systems, almost all UniVerse terminal input and output operations use an enhanced version of the UNIX terminfo facility. On Windows platforms, terminfo is part of UniVerse and is not used by the operating system. The terminfo facility lets programmers construct hardware-independent applications. Terminal I/O need not depend on terminal-specific control sequences.

The system administrator can define terminal-dependent character sequences for terminal operations such as clearing the screen, assigning an erase character, positioning the cursor, and deleting characters. These are defined in a set of files called the terminfo library. Programs can access this library during execution and select the proper control sequences based on the terminal type defined by the UniVerse `SET.TERM.TYPE` command. This is how ReVise positions field prompts and how

the UniVerse run machine implements the `PRINT @(column,row)` functions for different terminal types.

UniVerse files

Every UniVerse file logically comprises at least one data file and an associated file dictionary. Data files contain records that store data values in fields. File dictionaries contain records that define the contents of data files, as well as the way data is processed and displayed. Most UniVerse files have both a dictionary and at least one data file, although a dictionary can be created that has no data files associated with it, and a data file can be created that is not associated with a dictionary.

An entry in the VOC (vocabulary) file defines a UniVerse file. It defines the relationship between a data file and its associated dictionary. Records in the file dictionary do the following:

- Define fields in the associated data file
- Define different views of data stored in the same data field
- Process data stored in fields
- Translate data from other files
- Define output specifications and report formats

Records stored in UniVerse files are of variable length; so are the fields that make up the records. The only limit to the number of records that can be stored in a file is the size of your hard disk. Each record is identified by a unique key called the record ID.

UniVerse file structure

UniVerse files are implemented using system directories and files. UniVerse provides several kinds of file organization. This variety of storage methods simplifies application design and provides superior performance.

The following file structures are available:

- Type 1 and type 19 files (nonhashed)
- Static hashed files
- Dynamic files
- B-tree files

A data file can be either nonhashed or hashed, depending on the kind of data you want to store. File dictionaries are usually hashed.

Nonhashed files

Type 1 and type 19 nonhashed files store text, program source code, or other data that does not have much structure to it. These nonhashed files are implemented as directories in which each record is stored as a separate file.

Hashed files

Hashed files allow rapid access to individual records in a file, regardless of the number of records in the file. This is due to the way the records are stored and retrieved.

A hashing algorithm uses the record ID to distribute records in one or more groups. This algorithm generates the address of the group where the record is stored. To find a record in a hashed file,

UniVerse locates a group address on the disk. Within that group, individual record IDs are examined to identify the record.

Dynamic files

Dynamic files automatically adjust the number of groups as the amount of data in them changes. Dynamic files let you manage them more easily. You can increase file efficiency of some dynamic files by changing default parameters.

B-Tree files

B-tree files differ from other UniVerse hashed files. B-tree, or balanced-tree, files store records in sorted order. This reduces search time and uses a less costly record retrieval operation for secondary index files.

The XDEMO account

Starting at UniVerse 11.3.1, you can use the XDEMO account to test and use UniVerse commands. The XDEMO account is a group of test files that are installed automatically with UniVerse on Windows. For UNIX platforms, the XDEMO account is optional.

Note: XDEMO does not include any indexes; only test programs and files are included.

For more information about the installation of XDEMO, see the *Installation Guide*.

XDEMO files

The following table describes the files in the XDEMO account in more detail.

File	Description
FUR_REV	Contains 512 records, it includes an example revenue file.
LOCATIONS	Contains 3 records. Addresses, phone and fax numbers are not real numbers.
MEMBERS	Contains 2500 records. It includes imaginary credit card information and other data. The data is generated randomly. For the PASSWORD and CARDNUM fields, there is an additional ENC.PASSWORD and ENC.CARDNUM field that contain encrypted versions of those fields. In the dictionary of the MEMBERS file, a record called ENC.PARAMS reports which parameters were used for these fields. In the XDBP directory, there is a DECRYPT.EXAMPLE program that shows how the data can be decrypted again for passwords and credit card numbers. Records in this file contain no real life data.
PLOCATION	Contains 57399 records. This includes all delivery pizza locations in the USA. It includes longitude and latitude positions for use with mapping software.
PP & PBP	Contains U2 Python code examples, PP is just Python code and PBP is BASIC code just for Python examples.
PRODUCT_REVIEW	Contains 2 records only.

File	Description
PRODUCTS	Contains 390 records. There are no duplicate editions of titles, and only movies that have a synopsis are included.
REGION	Contains about 3 records, all related to the PLOCATION file.
RENTAL_DETAILS	Contains 2240 records. The dictionary items have similar syntax.
STATES	Includes a REGION and DIVISION field.
STATES_MAPS	Contains 54 records. Used in conjunction with PLOCATION.
XDBP	Basic programs used in examples. **Note:** The programs in the XDBP/XDBP.O files are cataloged using the DIRECT option on UniData and LOCAL on UniVerse.
WORLD_MAP	Contains 239 records. Related to PLOCATION file.
ZIPCODES	Contains 100 records. Related to PLOCATION file.

VOC file

Each UniVerse account has a vocabulary file called the VOC file. The VOC file contains entries that identify every verb, sentence, paragraph, file, keyword, and menu that you can use while you are in a UniVerse account. The command processor uses the VOC file to decide what action to take when you enter a command.

When a new UniVerse account is created, a standard VOC file is copied from the system administrator's account to the new account. You can tailor the copied VOC file to the specified purposes of the account by adding synonyms for standard verbs and keywords, adding new file entries, or storing often-used sentences and paragraphs. UniVerse files defined in an account's VOC file can be in the same directory as the VOC file, or they can be in other UniVerse account directories. UniVerse provides commands for setting up new VOC entries to access UniVerse files in other UniVerse accounts and for switching to other UniVerse accounts.

UniVerse commands

When you enter the UniVerse environment, the command processor displays a prompt (>). You can enter any UniVerse command at this prompt.

A UniVerse command line is called a sentence. It is like an imperative sentence in English. The first word of a sentence is always a UniVerse verb (such as `LIST or SELECT`) or the name of a stored sentence, a stored paragraph, or a menu. The other words in the sentence can be file names, record IDs, keywords, field names, phrase names, or values. The command processor either executes the command or passes it to another UniVerse or system process for processing.

For example, when you enter the following sentence, the UniVerse command processor looks up `LIST` in the VOC file:

```
>LIST PAYABLES WITH PYMT.DATE AFTER "8/1/91"
```

Since `LIST` is a RetrieVe verb, control is passed to the RetrieVe processor to execute the sentence.

You can store any sentence for future use by creating a stored sentence record in the VOC file. You can also store a sequence of sentences by creating a paragraph entry (for example, a record). Stored sentences and paragraphs let you repeat an operation or a sequence of operations often.

A stored sentence or paragraph can also use parameters specified on the command line or supplied in response to a prompt. This makes it easy to write generalized commands that execute differently, depending on the supplied parameters.

Review of terms

This chapter introduces the main components that make up the UniVerse system. Subsequent chapters describe these components in more detail. The following table summarizes the more important terms used in this chapter:

Term	Definition
B-tree file	A file that stores records in sorted order for ease of access.
command processor	The UniVerse processor that interprets command lines (UniVerse sentences) and either executes them or passes them to the appropriate UniVerse or system process or utility. When other processing completes, control usually returns to the command processor.
data file	A file, usually associated with a dictionary, containing records that store data values in fields.
database	A collection of logically related data that is organized for efficient storage and retrieval. It can be a single file or it can comprise a group of related files.
dictionary	A file that defines the contents and structure of at least one data file that is associated with it.
dynamic file	A file (type 30) that automatically adjusts the number of groups as the amount of data stored in them changes.
entry	A record in a file dictionary (dictionary entry) or in the VOC file (VOC entry) of an account.
field	A logical subdivision of a record that can contain data values.
group	A logical area of disk storage for storing records in a hashed file.
group buffer	A physical block of disk space for storing records in a hashed file.
hashed file	A file that uses a hashing algorithm for distributing records in one or more groups on disk. The algorithm is applied to each record ID to generate the address of the group where the record is to be stored.
keyword	An element of a UniVerse sentence that modifies the action of the verb. Arithmetic, relational, and logical operators are also keywords.
nonhashed file	There are two types of nonhashed file in UniVerse: B-tree files and files that are implemented as directories. The latter are known as type 1 and type 19 files. Each record in a type 1 or type 19 file is stored as a separate system file in the directory that serves as the UniVerse file.
paragraph	A sequence of UniVerse sentences (command lines) stored as an entry in the VOC file of an account. When the name of the paragraph is entered at the UniVerse prompt, the sequence of stored sentences is executed.
phrase	Any part of a RetrieVe sentence, excluding the verb. Phrases can be stored in any file dictionary or in the VOC file of an account. When a phrase name is used in a sentence, the name is expanded to the full contents of the phrase.

Term	Definition
proc	A stored procedure that defines a sequence of operations to be performed by the ProVerb processor. Procs are similar to shell scripts in the operating system environment and to job control languages on some mainframes.
ProVerb	The UniVerse processor that interprets and executes command statements stored in a proc.
record	A collection of related data values stored as a separate item in a UniVerse file. Every record has a record ID or key, and can comprise one or more fields containing data values.
record ID	The key used to gain access to a record in a file. Each record ID must be unique in any file.
RetrieVe	The UniVerse database query processor and report writer. RetrieVe commands select records, process data in them, and format the data for reports displayed on the screen or sent to the printer.
ReVise	A dictionary-driven data entry processor used for adding, changing, and deleting data in UniVerse files.
run machine	The UniVerse processor that executes compiled UniVerse BASIC programs.
sentence	A complete UniVerse command line. All sentences must begin with a UniVerse verb.
stored sentence	A sentence stored as an entry in the VOC file of a UniVerse account. When the name of the stored sentence is entered at the UniVerse prompt, the sentence itself is executed.
terminfo	A database containing descriptions of the capabilities of many different kinds of terminal. *terminfo* enables application programs to work with a variety of terminals without needing different codes.
UniVerse account	Working environment defined by a VOC file and all its related files. When users invoke the UniVerse environment, they initiate a UniVerse session in the UniVerse account defined by the VOC file in the current working directory.
UniVerse BASIC	The UniVerse procedural language processor and programming language.
UniVerse file	A UniVerse file that logically comprises a file dictionary and at least one data file. The relationship between the dictionary and the data file is defined by an entry in the VOC file.
verb	The action word (such as `LIST or COPY`) in a UniVerse sentence that initiates processing. All UniVerse sentences must begin with a verb.
VOC file	The master file in a UniVerse account. The VOC file contains records that identify all verbs, sentences, paragraphs, files, keywords, procs, and menus that you can use when you are logged in to the UniVerse account.

Chapter 2: UniVerse accounts

The first part of this chapter describes UniVerse accounts. The second part is for UNIX users only, and describes the relationship between UniVerse accounts and UNIX accounts. In this manual, login accounts are called user accounts or login accounts, and accounts in the UniVerse environment are called UniVerse accounts.

Your UniVerse account

On UNIX systems, depending on how the system administrator sets up your login account, you can log on to the UNIX shell environment and then enter `uv` to access UniVerse, or you can log on directly to the UniVerse environment. On Windows platforms, you always enter UniVerse from a telnet session.

A UniVerse account is located in a directory containing a set of special UniVerse files. The most important UniVerse file is the VOC file, which contains the vocabulary of that account.

Windows UniVerse accounts

You can set up UniVerse accounts to use short user names or fully-qualified user names. You can also set up a local logon policy for UniVerse users.

User names and domain names

Before Release 9.3.1, UniVerse on Windows platforms used short user names that did not include the domain name. As of Release 9.3.1, fully-qualified user names are used in the format:

```
DOMAIN.NAME\username
```

If you want to use short user names, use `regedt32` to add an entry to the registry on platforms other than Windows 7 and Windows 2008 as follows:

1. Edit the Windows registry using the `regedt32` command. Choose one of the following:
   - For 32-bit systems: HKEY_LOCAL_MACHINE -> Software -> ROCKET SOFTWARE -> UniVerse -> CurrentVersion
   - For 64-bit systems: HKEY_LOCAL_MACHINE -> Software ->Wow6432Node-> ROCKET SOFTWARE -> UniVerse -> CurrentVersion
2. From the **Edit** menu, choose **Add Value**. The Add Value dialog box appears.
3. In the **Value Name** field, enter `UseShortUserNames` . The Value Name is case-sensitive.
4. In the **Data Type** field, choose **REG_DWORD**. The DWORD Editor dialog box appears.
5. In the **Data field**, enter `0` for long user names, or enter `1` for short user names.
6. Under **Radix**, click **Decimal**, then click **OK**.

On Windows 7 and Windows 2008, complete the following steps:

1. Edit the Windows registry using the `regedt32` command. Choose one of the following:
   - For 32-bit systems: HKEY_LOCAL_MACHINE -> Software -> ROCKET SOFTWARE -> UniVerse -> CurrentVersion
   - For 64-bit systems: HKEY_LOCAL_MACHINE -> Software ->Wow6432Node-> ROCKET SOFTWARE -> UniVerse -> CurrentVersion
2. From the **Edit** menu, choose **New**, then choose **DWORD**. A new DWORD is created with focus on the name.
3. Enter `UseShortUserNames` as the **DWORD** name.

4. Once created, right-click **DWORD** and choose **Modify**.
5. In the modification dialog box, select **Decimal** and set the Decimal value to `1`.
6. Under **Radix**, click **Decimal**, then click **OK**.

> **Note:** If you configure UniVerse to use short user names, you cannot use the `MESSAGE` command to send messages to all users on the local domain, as the domain name is not present in the user name.

A domain user and a local user with the same user name are treated as different users. For example <*domain name*>*alice* and <*hostname*>*alice* are not recognized as the same user. This affects the `SQL GRANT` statement. If your users can log on both locally and from a domain, you must specify both the domain and the local user name in the `GRANT` statement. The local user name must include the hostname. For example:

```
>GRANT CONNECT TO mymachine\alice, SALES\alice;
```

mymachine is the hostname, *SALES* is the domain name.

Local logon policy

After UniVerse is installed on a Windows system, only members of the Windows Administrators group can log on to UniVerse. To allow users other than administrators to log on to UniVerse, you or your system administrator must assign local logon permissions for them. See your Windows documentation for details.

UniVerse account flavors

Any UniVerse account can be one of several standard *flavors*: IDEAL, IN2, INFORMATION, PIOPEN, PICK, or REALITY. New users are encouraged to select the IDEAL UniVerse flavor. The IDEAL flavor contains the best of both the Pick and Prime worlds.

The INFORMATION flavor maintains an environment compatible with Prime INFORMATION systems. The PIOPEN flavor is compatible with PI/open. The REALITY, PICK, and IN2 flavors are compatible with different versions of the Pick system. These flavors can be chosen by users who have experience with Prime or Pick systems and want UniVerse to operate the same way.

UniVerse login entry

In the UniVerse environment, the system administrator can set system-wide defaults by placing appropriate commands in a paragraph, sentence, proc, menu, or a UniVerse BASIC program named `UV.LOGIN` in the VOC file of the UV account. The `UV.LOGIN` entry is executed every time a user logs on to any UniVerse account.

You can also create a local login entry in the VOC file of any UniVerse account. The login entry can be a paragraph, sentence, proc, menu, or UniVerse BASIC program. The login entry is executed after the `UV.LOGIN` entry when a user logs on to that particular UniVerse account. The record ID of a UniVerse account's login entry depends on the flavor of the account. In IDEAL, PIOPEN, and INFORMATION flavor accounts, a login entry is always called LOGIN. In PICK, IN2, and REALITY flavor accounts, the login entry can be one of the following:

- The user's login name

- The name of a UniVerse account, read from the UV.ACCOUNT file in the UV account. The UV.ACCOUNT file defines all UniVerse accounts on the system
- LOGIN, the name of a record in the VOC file of the UniVerse account in which you are currently working

UniVerse looks for a login entry in the order shown.

ON.EXIT entry

You can create an ON.EXIT entry in the VOC file to ensure that certain commands are executed each time you exit UniVerse. Depending on how you exit UniVerse, your ON.EXIT entry is executed differently, as summarized:

Exit method	ON.EXIT execution
Leave UniVerse	When you enter QUIT in UniVerse, the ON.EXIT entry is executed.
Log out	If your ON.EXIT entry contains a logout verb, you log out when you enter QUIT.
UniObjects (UCI)	Not executed by this process type.

ON.ABORT entry

The ON.ABORT entry in the VOC file is executed when a program aborts. You can create the ON.ABORT entry to ensure that certain commands are executed after an abort or to rerun the paragraph during which the abort occurred. The ON.ABORT entry will not be invoked if the abort occurs from a UniObjects (UCI) process.

If the ON.ABORT entry does not exist, you return to the system prompt (>) when a program is aborted.

Creating a UniVerse account

The recommended way to create a UniVerse account is to have the system administrator set it up for you.

However, if you use the uv command in a directory that has not been set up for a UniVerse account, you are prompted to set up the directory as a new UniVerse account. If your system administrator has not restricted you to a particular flavor, you are also prompted to choose an account flavor.

Updating your account

The RELLEVEL entry in your VOC file contains the current release level of your account. Each time you log on to the account, this entry is checked to make sure your account is up to date. If your account is not current, you are prompted to update it. To display the RELLEVEL entry, enter .L RELLEVEL at the system prompt.

You can also use the UPDATE.ACCOUNT command to update your account or change the flavor of your account. UPDATE.ACCOUNT works only on changes made within the current release level.

While your VOC file is being updated, records that are being replaced are moved to the &TEMP& file to prevent them from being destroyed. The names of records moved to &TEMP& are listed on your screen.

UNIX and UniVerse account differences

A UNIX login account and a UniVerse account are different. In the UNIX environment, each user is generally given an individual login name, often called the user's UNIX account, which includes a home directory under which the user can create his or her own private files. In UNIX a login account is defined by a line in the UNIX /etc/passwd file that specifies the user's login name, password, home directory, and login shell. Login accounts in UNIX are like personal working environments that stay with users no matter where they are working on the system.

Once UNIX users log on to the system, they have access to all directories and files on the system, except those protected by file permissions. Users can change their current working directories without changing other aspects of their working environment. Files and commands in other directories can be accessed by entering the path identifying the location in the file system's complete directory tree.

In the UniVerse environment the user's working environment is determined primarily by the UniVerse account in which the user is currently working. UniVerse accounts are more self-contained than UNIX login accounts. The VOC file contained in each UniVerse account defines the account environment, including all the files and all the commands that are available to users who are logged on to the account.

Different environments for different users

Some sites make both environments available to users and create a separate UniVerse account for each user. At other sites users may work primarily in the UniVerse environment, and a group of users may work in a particular UniVerse account. For example, a UniVerse account might be defined for a department rather than for an individual. Each user of the SALES account might be given his or her own login name at the UNIX level but be assigned the same home directory and share the same UniVerse account.

Your UNIX account

A UNIX login account gives a user access to the UNIX system and a place to store files. The login account does not limit access to other directories. It is often necessary to access information stored in other directories, such as the UNIX commands themselves.

The system administrator gives each user a UNIX login ID and a password. Users also have a home directory in which they can store files and create subdirectories.

The standard UNIX shell environment

The standard UNIX environment presents each user with a shell, or command interpreter, at login time. The shell puts the commands that you enter at the system prompt into a form that the computer understands. Three types of shell are provided on most UNIX systems:

- Bourne shell (*sh*)
- C shell (*csh*)
- Korn shell (*ksh*)

Although UniVerse is a very large set of programs, it can be thought of as a UNIX shell. It takes input from you and performs operations based on that input. Like a UNIX shell, UniVerse has a number of commands that it understands.

The system administrator can set up login accounts so that users enter a shell at login time. In a UniVerse environment that shell is typically UniVerse itself. When you log out of your initial shell, you exit the system entirely. If you log directly in to UniVerse, you may never interact directly with the underlying operating system environment.

Moving from one UNIX shell to another

In the UNIX environment you can move from one shell to another. For example, you can move between the Bourne shell and C shell using the UNIX `sh` and `csh` commands. To access a UNIX shell from the UniVerse environment, you can use the UniVerse commands `SH` and `CSH`. When you exit a shell that you entered from another shell, you exit to the calling shell.

Note: If you invoke a UNIX shell from the UniVerse environment with the `SH` or `CSH` command, be careful to return to UniVerse either using the `exit` command or pressing Ctrl-D. Do not use the UNIX `uv` command.

Entering UniVerse from a UNIX shell

To enter UniVerse from a UNIX shell, enter the command `uv` at the UNIX shell prompt. When you use the `uv` command to invoke the UniVerse environment, you enter the UniVerse account that has been set up in the directory you presently occupy (known as your current working directory). If the directory has not been set up as a UniVerse account, a message prompts you to set up the directory as a new UniVerse account.

Shell environment initialization

In general, when a UNIX shell is initialized, it executes a sequence of commands stored in a special file in the user's home directory. Similarly, when a user invokes UniVerse, a sequence of commands stored in a record in the user's VOC file can be executed to initialize the UniVerse environment. These commands can modify the default UNIX or UniVerse environment for a specific UNIX or UniVerse account.

See UniVerse login entry, on page 20.

Bourne shell environment

In the Bourne shell environment, a file called `.profile` in the user's home directory is executed when a Bourne shell (*sh*) is executed as the login process.

C shell environment

In the C shell environment, the `.cshrc` and `.login` files in the user's home directory are executed when a C shell (*csh*) is executed as the login process. The `.cshrc` file is executed each time a new C shell is created.

Chapter 3: The command processor and the VOC file

This chapter describes the UniVerse command processor and how it uses the VOC file. It also describes the VOC file's facilities for storing commands and sequences of commands for future use.

The command processor

The command processor examines every line entered at the system prompt, as well as commands entered from a stored command sequence, a proc, or a UniVerse BASIC program.

The command processor parses a command and searches for the verb in the VOC file. Depending on the verb, the command processor either executes the command or calls the proper processor to complete the execution. The actions taken by the command processor and other processors depend on definitions in the VOC file.

The command processor also lets you store sentences and paragraphs (a sequence of sentences) in the VOC file for execution later.

The command processor maintains a list of the most recent command lines entered at the system prompt. This list is called the sentence stack. You can use the sentence stack to recall, delete, change, or reexecute a previous command, or to save a sentence or paragraph in the `VOC` file.

Special character interpretation

The UniVerse command processor recognizes special control characters that delimit fields, values, and subvalues in stored data as shown in the following table.

Control character	Description	Meaning	Value
Ctrl-^	The Control key and the Up Arrow (or caret) key	Field mark	^254
Ctrl-]	The Control key and the right bracket key	Value mark	^253
Ctrl-\	The Control key and the Backslash key	Subvalue mark	^252

Using a period and a hyphen in verbs and keywords

The command processor recognizes that a period and a hyphen can be used interchangeably. For example, if you enter `CREATE-FILE` and the command processor cannot find a VOC entry with that ID, it searches for CREATE.FILE.

Displaying other special function characters

Other characters that have special functions can be displayed by using the `PTERM` (UNIX) or `PTERM` (Windows Platforms) `DISPLAY` command. For example:

```
>PTERM DISPLAY
MODE         EMULATE
CC           INTR  = DEL QUIT SUSP = OFF DSUSP  = OFF
```

```
                        SWITCH = ^@ ERASE = ^H WERASE = OFF KILL = ^X
                        LNEXT = OFF REPRINT= OFF EOF = ^D EOL = ^@
                        EOL2 = ^@ FLUSH = OFF START = ^Q STOP = ^S
                        LCONT = ^_ FMC = ^^ VMC = ^] SMC = ^
INPUTCTL          ON
CARRIER           RECEIVE HANGUP -LOCAL
CASE          -UCIN -UCOUT -XCASE INVERT
CRMODE        -INLCR -IGNCR ICRNL ONLCR -OCRNL -ONOCR -ONLRET
DELAY    BS0 CR0 FF0 LF0 VT0 TAB0 -FILL
ECHO          ECHO ERASE=BSB KILL=LF CTRL -LF
HANDSHAKE     XON -ANY -TANDEM -DTR
OUTPUT        POST -TILDE -BG CS EXPAND
PROTOCOL      LINE=0 BAUD=9600 DATA=8 STOP=1 EVEN DISABLE STRIP
SIGNALS       ENABLE FLUSH BREAK=INTR
```

Using the interrupt, quit, and suspend keys

When you press the Intr, Quit, or Susp key, you are given a set of options to choose from to determine what the system should do next. There are five options available, but the D option is available only in a UniVerse BASIC program.

Option	Description
A	Aborts the current process.
C	Continues the current process.
D	Enters the debugger.
L	Aborts the current process and runs the login procedure.
Q	Aborts the current process and exits the UniVerse environment.

Note: At the `Press any key to continue...` prompt, the Q to quit is ignored when in BREAK OFF mode.

Phantom processes

You can run any command in the background provided it does not require terminal input. To run a process in the background, add the verb `PHANTOM` before the command. `PHANTOM` creates a record in a type 1 file called `&PH&` to store output resulting from the command. The record ID of the `&PH&` record is in this format:

command_time_date

command is the verb that immediately follows the word `PHANTOM` on the command line. *time* and *date* are the time and date when the command started to execute.

If the process requires terminal input, you must use DATA statements in a paragraph to supply the input.

For example, you can resize the EMPLOYEES file in the background:

```
>PHANTOM RESIZE EMPLOYEES DYNAMIC
```

When the process starts, a message such as the following appears:

`Phantom` process started as Process ID *pid#*

The operating system assigns the process ID number, *pid#*.

If you have reached the system-wide limit on the number of processes that can be running at any one time, a message such as the following appears:

```
NO FREE PHANTOMS
 You cannot run a phantom process now. Wait a while, then try
again.
```

To monitor `phantom` processes started during the current login session, enter `STATUS ME`.

When a `phantom` process finishes, UniVerse notifies you the next time you return to the UniVerse prompt. If you want to be notified immediately, use the `NOTIFY ON` command.

You can also stop a `phantom` process before it is finished. You can stop only `phantom` processes that you have started during the current login session. Enter the following:

`>LOGOUT` *pid#*

The *pid#* is the process ID number assigned when the process started. Use `STATUS ME` to get the *pid#* of the process you want to stop.

PHANTOM use that will consume a database license

The following UniVerse BASIC functions, which provide interactive capability when used in a `PHANTOM`, will consume a database license:

- UniVerse BASIC Socket API:
  - openSocket()
  - openSecureSocket()
  - initServerSocket()
  - initSecureServerSocket()
- UniVerse BASIC MQ API:
  - amInitialize()
- UniVerse BASIC CallHTTP:
  - submitRequest()
- UniVerse BASIC CallSOAP:
  - SOAPSubmitRequest()
- TIMEOUT() when using microsecond timeout.

The license will be released when the `PHANTOM` exits. If no license is available, the process fails with a status code 99, indicating that UniVerse failed to obtain a license for the interactive PHANTOM process.

The VOC file

Every word you use in a UniVerse sentence must be defined to UniVerse as one of the following:

- A record ID of a record in the file being processed by the command
- A constant representing a field value
- An entry in the dictionary of the file being processed by the command
- An entry in the VOC file of the account

Every UniVerse account has a VOC file containing a record for every verb, file name, remote pointer, sentence, paragraph, menu, and keyword that you can use while in the account. The VOC file can also

contain field definitions and phrases that are not specific to a particular file and thus are not included in a particular file dictionary. Records in the VOC file are also called VOC entries.

The VOC file is a data file and has a dictionary associated with it. The names of the verbs, file names, keywords, and other items defined in the VOC file are unique IDs of records in the data file. For example, LIST is the record ID for the verb LIST, and an equal sign (=) is the record ID for the arithmetic operator *equals*.

To store a record in the VOC file, you must give it a name. You can then use that name in a command line. The UniVerse command processor verifies that the entry is in the VOC file, then processes it according to the entry type.

Listing the VOC file contents

To look at the contents of the VOC file in the current account, use the `LIST` command as follows:

```
>LIST VOC
```

This command displays a list that looks like the following:

```
LIST VOC 03:03:11pm  05 Jan 1995 PAGE 1
NAME.....   TYPE    DESC.........................

GLOBAL.CATDIR    F       File - Used to access system
                 catalog space.
STAT         V       Verb - Produce the STAT of a
                 named field in a file
A        K       Keyword - The article "a";
                 ignored by RetrieVe
FIRST       K       Keyword - Limit the display of
                 lines in a report
HELP         V       Verb - Invoke the HELP
                 facility
THE        K       Keyword - The article "THE";
                 ignored by RetrieVe
UNIQUE     K       Keyword - Used in conjunction
                      with SAVING to save only
                 unique values.
VVOC         V       Verb - Verify VOC file against
                 NEWACC file, listing
                 differences.
DF.MODIFY        V       Verb - Modify a Distributed
                 File
DISKS       K       Keyword - Display STATUS
Press any key to continue...
```

VOC file record format

Every entry in the VOC file has a unique record ID and two or more fields. The dictionary of the VOC file, DICT VOC, contains records that define the names and formats of fields in the VOC file. The format of fields in the VOC file differs depending on the kind of record.

Every record ID field is defined in the VOC dictionary as NAME, KEYWORD, or F0, which are synonyms for the @ID record. The record ID field is sometimes called field 0, but it is not considered part of the VOC entry proper.

Field 1 (F1)

The first field of every VOC entry determines what kind of entry it is. Field 1 is defined in the VOC dictionary as F1. F1 contains the type code and can have two components: the type code itself and a brief description of the entry. The first one, two, or three characters in F1 must be one of the following:

Code	Type of VOC entry
D	Data field definition
F	File pointer
I	I-descriptor definition
K	Keyword
M	Menu selector
PA	Paragraph
PH	Phrase
PQ	ProVerb (proc)
PQN	ProVerb (proc)
Q	Remote file pointer (Q-pointer)
R	Remote command pointer
S	Stored sentence
V	Verb
X	User record

The rest of F1 is available for an optional description of the record. F1 for a file pointer might look like this:

```
F Accounts payable file for this account
```

Fields 2 through 5 (F2, F3, F4, F5)

The VOC dictionary defines fields 2 through 5 of the VOC file as F2, F3, F4, and F5. The dictionary also includes definitions that serve as synonyms for these fields. The definitions vary for different kinds of data. For example, if the VOC entry is a file name, field 2 contains the path for the data file and field 3 contains the path for the file dictionary.

Use the field definitions when you want to refer to the data in VOC entries that define UniVerse files. The following examples show the VOC dictionary entries defining F2 (field 2) and F3 (field 3) with their synonyms `FILE.NAME` (field 2) and `DICT.NAME` (field 3). The first column shows the line numbers of the VOC entries. Line numbers are not part of the entries.

```
            F2          FILE.NAME           F3          DICT.NAME
0001: D       I              D      I
0002: 2       F2             3      F3
0003:
0004:         File Name             DICT Name
0005: 13T     15T           13T     15T
0006: S       S             S       S
0007:
```

F2 and F3 are generic field definitions that describe any data stored in fields 2 and 3. `FILE.NAME` and `DICT.NAME` are specific field definitions describing the data stored only in type F VOC entries.

Additional fields

Fields after field 5 are generated by UniVerse and contain information such as file names of multiple data files associated with the same file dictionary. The following table shows the record formats for 11 VOC entry types. Information shown in brackets is optional.

	VOC entry type				
Field	**Verb**	**Remote command entry**	**Stored sentence**	**Paragraph**	**File definition**
F1	**V** [*desc*]	**R** [*desc*]	**S** [*desc*]	**PA** [*desc*]	F [desc]
F2	*processor*	*file name*	*sentence*	*sentence 1*	*O/S path for data file*
F3	*dispatch type*	*record ID*	[*continuation of sentence*]	*sentence 2*	*O/S path for file dictionary*
F4	*processor mode*	[*security subroutine*]		. . .	[**M**]
F5	reserved			. . .	[*O/S path for Pick dictionary*]
F6	reserved			. . .	[*subitem field*]
F7	reserved				[*file names, multiple data files*]
F8	reserved				[*O/S path, multiple data files*]

	VOC entry type					
Field	Q-Pointer	Keyword	Phrase	Menu	User record	ProVerb
F1	**Q** [*desc*]	**K** [*desc*]	**PH** [*desc*]	**M** [*desc*]	**X** [*desc*]	**PQ** [*desc*]
F2	*account*	*operation number*	*phrase text*	*file name*	*user-supplied information*	*proc statement 1*
F3	*file name*	[*sentence*]	[continuation of phrase	*record ID*	. . .	*proc statement 2*

VOC entry types

This section describes the contents of each field in a VOC entry for each record type except for D- and I-descriptors. The field number and the information contained in the field are listed. Optional information is in brackets.

For more information, see UniVerse file dictionaries, on page 63.

The VOC file contains six entry types that define commands or point to verbs or other commands stored in other accounts or files. These entries define verbs, remote command entries, stored sentences, paragraphs, procs, and menus.

V: Verb

A verb is a UniVerse command. The VOC entry for a verb specifies the processor that the verb invokes, the dispatch type, and the flags that the processor uses. The record ID or name is the verb itself. To display all the verbs in the VOC file, use the `LISTV` command.

A VOC verb entry contains the following:

Line	Field contents
001:	`V` [*description*]
002:	*processor*
003:	*dispatch type*
004:	*processor mode*
005:	reserved
006:	flavor
007:–*nnn*:	reserved

description is a brief explanation of the verb's use. *processor* is the name of the program that is executed.

dispatch type can be one of the following codes:

Code	Dispatch type
`B`	BASIC program
`C`	C shell script
`D`	DOS batch files
`E`	External
`I`	Internal
`P`	Primitive command
`Q`	Query command
`S`	Bourne shell script
`U`	Operating system command

processor mode is used by the command processor to perform the required setup before executing the verb. It can be one or more of the following codes:

Code	Processor mode
`A`	Alternate query syntax
`B`	*not used*
`C`	External program COMO
`D`	Pass DATA to subprocess
`E`	Use Win32 expansion routine (Windows NT only)
`F`	Pass format via environment variable
`G`	Allowed in an SQL CALL statement
`H`	`EXECUTE` and `PERFORM` can use in a transaction
`I`	Interrupt control
`K`	Keep select list
`M`	Maps `EXECUTE` output to internal character set (NLS only)

Code	Processor mode
N	Does not set @SYSTEM.RETURN.CODE
P	Parenthetical options are available
Q	SQL mode
R	Backslashes (\ ) can be used to quote strings
S	Use select list
T	Terminal mode change
U	Add path of the UV account directory before processor name
V	Function specified by field 5
X	Read DATA if a select list is active
Y	Suppresses recording DAT.BASIC.* event entries during audit logging for commands that call BASIC programs. This is the default functionality. Note that this option cannot be used to bypass BASIC event logging in user-created VOC entries.
Z	Clear the VOC cache.

flavor specifies the UniVerse flavor in which the verb executes. By default, verbs execute in the flavor of the current account.

R: Remote command entry

A remote command entry is an entry in the VOC file that points to a command stored as a record in another file.

By putting the remote command entry in the VOC file, you can store larger command records such as paragraphs or procs in another file (such as the VOCLIB file) and keep the VOC file from becoming unwieldy. Storing larger records of different sizes in files other than the VOC keeps VOC file records relatively uniform in size, which improves performance. You might store all the records that pertain to an application in a single file and simply point to them from the VOC file.

You can also use a remote command entry to restrict access to a command. The remote command entry in the VOC file specifies the UniVerse file name and the record ID of the remote command. To display the remote command entries in the VOC file, use the LISTR command.

Note: Some UniVerse commands require certain verbs to be in the VOC file. Moving such verbs to another file may render some UniVerse commands inoperable.

A remote command VOC entry contains the following:

Line	Field contents
001:	R [*description*]
002:	*filename*
003:	*record.ID*
004:	[*security subroutine*]
005:	Y

filename is the name of the UniVerse file where the record is stored. *record.ID* is the actual record. *security subroutine* is optional. It is a subroutine that is executed before accessing the remote command. It restricts access to the remote command.

The `Y` *audit mode* option suppresses recording `DAT.BASIC.*` event entries during audit logging for commands that call BASIC programs. This is the default functionality. Note that this option cannot be used to bypass BASIC event logging in user-created VOC entries.

Security subroutines

You can use a security subroutine to evaluate users' access to remote commands. A security subroutine sets a flag permitting or restricting access to the remote command. The UniVerse command processor checks the flag returned by the subroutine before accessing the remote command. The syntax is as follows:

SUBROUTINE *security (remote, sentence, level, port, acct, user, flag)*

security specifies the name of the subroutine. A security subroutine must have seven arguments. The first six arguments are passed to the subroutine by the command processor. The last argument is a return argument. The arguments are as follows:

Argument	Description
remote	Contents of the remote command entry itself
sentence	The value of `@SENTENCE` (that is, the command that invoked the remote command)
level	The following numeric values are set: 0 Specifies the command processor 1 Specifies execute 2 Specifies execute of execute
port	User's port number
acct	Current account name
user	Login name of the user
flag	Returns one of the following numeric values: 1 Permits access 0 Restricts access

Upon return from the subroutine, the command processor checks the return flag. If the flag is set to 1, the remote command is executed. If the flag is set to 0, access to the remote command is denied.

Security subroutine example

Suppose that you do not want anyone to run the `RESIZE` command during normal business hours. Here is a security subroutine that restricts the execution of any command to off-hours unless the user is logged on as *su* (the superuser):

```
SUBROUTINE TIME.LOCK (remote,sentence,level,port,acct,user,flag)
 *
 * If time of day is between 9 AM and 5 PM do not allow routine
 * to be executed unless user is su.
 *
 now = TIME()
 IF now > 32400 AND now < 61200
 THEN
            IF user = 'su' THEN flag = 1 ELSE flag = 0
       END
 ELSE
       flag = 1
```

```
END
```

If the subroutine called `*TIME.LOCK` is cataloged, use the following steps to implement it for the `RESIZE` command:

1. Copy the `RESIZE` command to a remote file, `VOCLIB`.

   ```
   >COPY FROM VOC TO VOCLIB RESIZE
   1 record copied.
   ```

2. Replace the old VOC entry for `RESIZE` with a new entry that points to the remote command. The new entry contains the following lines (shown as it appears in the UniVerse Editor):

   ```
   0001: R
   0002: VOCLIB
   0003: RESIZE
   0004: *TIME.LOCK
   ```

   Now, unless a user is logged on as *su*, or it is after hours, the `*TIME.LOCK` subroutine returns a value of 0 and does not permit access to the `RESIZE` command.

   ```
   >RESIZE SUN.MEMBER
   Security access denied.
   ```

S: Stored sentence

A sentence is a complete command line, including the verb. A sentence can include a file name, field names, selection and sort expressions, and keywords.

You can store sentences using the sentence stack `Save` command (`.S`) or using the Editor. If you use the Editor, be sure to identify the VOC entry as a sentence by putting type code `S` in field 1. To examine the sentences in the VOC file, use the `LISTS` command.

If you often use the same sentence and want to avoid typing it each time, save the sentence as an entry in the VOC file. A stored sentence can be one of the following:

- A complete sentence
- The name of another stored sentence
- The name of a paragraph
- The name of a menu

The record ID is the sentence name. A VOC sentence entry contains the following:

Line	Field contents
001:	`S` [*description*]
002:–*nnn*:	*sentence text*

Use the Editor to create a VOC sentence entry or use the sentence stack `command.S` to save a sentence. To use `.S`, use the following syntax:

`.S` *`name line#`*

name is the name you give the sentence. *line#* is the line number of the sentence in the stack that you want to save.

The `.S` stack command creates a VOC entry for the sentence with the record ID specified by *name* in the `.S` command line.

You invoke a sentence by entering its name at the system prompt. Command line arguments entered after the sentence name are appended to the sentence before it is executed.

PA: Paragraph

A paragraph is a series of sentences stored together under one name. Paragraphs let you execute several commands by entering the name of the paragraph at the system prompt.

The VOC paragraph entry contains the sentences that make up the paragraph. The paragraph can also contain special control statements and inline prompting to request input from a user when the paragraph is executed. BASIC programs can pass values for inline prompts to paragraphs with DATA statements. The record ID is the paragraph name.

The VOC paragraph entry contains the following:

Line	Field contents
001:	PA [*description*]
002:	*sentence 1*
003:	*sentence 2*
.	
.	
.	
nnn:	*last sentence*

To use `.S` to save a paragraph, use the following syntax:

.S *name start end*

name is the name you give the paragraph. *start* is the number of the sentence in the stack that you want to begin the paragraph. *end* is the number of the sentence in the stack that you want to end the paragraph.

UniVerse executes each sentence in order, just as if each command were entered at the keyboard. To examine the paragraphs stored in the VOC file, use the `LISTPA` command. Use the Editor to create a paragraph entry in the VOC file, or use the sentence stack command `.S`.

Note: When you specify the start and end numbers in the `.S` sentence stack command, UniVerse saves the higher number as the first sentence and the lower number as the last sentence, regardless of the order in which you enter them in the command line. For example, `.S name 5 2` is the same as `.S name 2 5`.

```
<.L
06 INITIALIZE.DEMO
05 LIST DICT ORDERS
04 LIST DICT CUSTOMERS
03 LIST DICT INVENTORY
02 LIST DICT VOC
01 LIST VOC
>.S STACKA 5 2
>.L STACKA
          STACKA
001 PA Saved at 11:22:06 05 Dec 1994 by userx
002 LIST DICT ORDERS
003 LIST DICT CUSTOMERS
004 LIST DICT INVENTORY
```

```
 005 LIST DICT VOC
>.S STACKB 2 5
>.L STACKB
                STACKB
 001 PA Saved at 11:22:41 05 Dec 1994 by userx
 002 LIST DICT ORDERS
 003 LIST DICT CUSTOMERS
 004 LIST DICT INVENTORY
 005 LIST DICT VOC
```

The `.S` stack command creates a VOC entry for the paragraph with the record ID specified by *name* in the `.S` command line.

F: File definition

UniVerse uses two kinds of VOC entry for defining files: F-descriptors and Q-pointers. F-descriptors can point either to local files stored in the same UniVerse account as the VOC file or to remote files stored in other UniVerse accounts.

The VOC F-descriptor entry specifies the paths of the file dictionary and all associated data files. The record ID of the entry is the UniVerse file name.

The VOC file descriptor entry contains the following:

Line	Field contents
001:	F [*description*]
002:	*O/S path of the data file* (usually the same as the UniVerse file name)
003:	*O/S path of the file dictionary* (usually the same as the UniVerse file name with the D_ prefix)
004:	[M]
005:	[*O/S path for Pick-style dictionary*] (usually the same as the UniVerse file name with the P_ prefix)
006:	[*subitem field*]
007:	[*UniVerse file names of multiple data files*]
008:	[*O/S paths of multiple data files*]

M (in 004) is an optional entry. It indicates that the UniVerse file comprises multiple data files. *subitem field* (in 006) uses the WITHIN keyword and specifies the field number. For more information, see the *UniVerse Guide for Pick Users*.

You can use any of these `LIST` commands to display files:

Command	Description
`LISTF`	Displays all the files defined in your VOC file, whether local or remote.
`LISTFL`	Displays only the files that are stored in your account.
`LISTFR`	Displays all the remote files that are defined in your VOC file.

VOC file pointers

When you use a file name in a UniVerse sentence, the file name refers to a VOC entry for the file. The VOC entry in turn points to the data file and the file dictionary. Usually the VOC entry contains

dictionary and data file paths that are generated from the name of the VOC entry, but a VOC entry can point to any dictionary or data file regardless of the name of the VOC entry.

Creating synonyms

You can create a synonym for a file by creating another VOC entry that points to the same dictionary and data files. This lets you use different names to refer to the same file. For example, you might have a file with a long name called `ACCOUNTS.RECEIVABLE.1995`. You can create a synonym called `AR.95` as a shorter name that is faster to type.

You can also create synonyms to associate several file dictionaries with one data file or a single dictionary with several data files. For example, you might have a data file called EMPLOYEES with its associated dictionary called D_EMPLOYEES. Suppose you want to use another dictionary called D_EMPL.PAYROLL with the EMPLOYEES data file. To do that, create a synonym that specifies D_EMPL.PAYROLL (instead of D_EMPLOYEES) in field 3. Call the synonym EMPL.PAYROLL.

In this example, the VOC file entry for EMPL.PAYROLL looks like the following:

```
     EMPL.PAYROLL
0001 F Synonym for EMPLOYEES file using EMPL.PAYROLL dictionary
0002 EMPLOYEES
0003 D_EMPL.PAYROLL
```

Creating file pointers

You can also create file definitions in the VOC that point to files in other UniVerse accounts.

You can point to a remote file either directly, using a full or relative path, or through the VOC file of the remote account using a Q-pointer.

The paths in fields 2 and 3 of remote file pointers are the full paths of the files. A remote file pointer must be changed if you change the location of the account where the file is located. For this reason a Q-pointer, which points indirectly to the remote file, is more portable than a remote file pointer. Provided the system administration files are updated properly, the Q-pointer continues to point correctly to the remote file even if the account is moved to a different directory.

You must have the proper access authorization for such files in order to use them. Use the `SETFILE` command or the Editor to create an entry that uses paths. Use the `SET.FILE` command (with a period) to create Q-pointers.

Q: Q-pointer

A Q-pointer points to another file descriptor either in the local VOC file or in the VOC file of another account. The VOC Q-pointer entry contains the following:

Line	Field contents
001:	Q [*description*]
002:	[*account*]
003:	*filename*

account is the name of a remote UniVerse account. If *account* is blank, the file descriptor pointed to is assumed to be in the local VOC file. *filename* is the record ID of the file descriptor in the VOC file of *account*.

K: Keyword

A keyword defines an operation that is to take place in the sentence or modifies the action of a verb. Some examples of keywords in UniVerse are =, GREATER, >, and WITH. The VOC file entry for a keyword specifies the internal operation number for that keyword. The record ID is the keyword itself. A keyword VOC entry contains the following:

Line	Field contents
001:	K [*description*]
002:	*operation number*
003:	[*sentence*]

operation number is an identifier that points to the code that performs the operation. *sentence* is optional. It is the stored sentence that is executed when the keyword is used as a verb (that is, when the keyword is the first word on the command line).

To see all the keywords stored in the VOC file, use the `LISTK` command.

Creating keyword synonyms

You can create a synonym for a keyword by creating a record in the VOC file that contains the same information as the original keyword. You can then use the new keyword in a sentence or phrase and get the same result as the original keyword.

PH: Phrase

A phrase is a part of a RetrieVe or ReVise sentence, very much like a phrase in an English sentence. Phrases can contain any element of a RetrieVe sentence except the verb.

The VOC entry for a phrase contains words that make up the phrase. The record ID is the phrase name. Usually phrases are stored in the dictionary of the file in which they are to be used. You may, however, want to define phrases in the VOC file so they can be used with several files in the same account. The phrase VOC entry contains the following:

Line	Field contents
001:	PH [*description*]
002:–*nnn*:	*the phrase*

To use a phrase in a sentence, include the phrase name anywhere after the verb.

Upon executing a RetrieVe sentence that specifies a file name and contains a phrase, UniVerse looks for the phrase in the specified file's dictionary first. If the phrase is not found in the dictionary, UniVerse examines the VOC file. UniVerse uses the contents of the phrase when it executes the sentence.

Use the Editor or ReVise to create a phrase entry in the VOC file.

To see all the phrases stored in the VOC file, use the `LISTPH` command.

M: Menu

A menu is a record in a UniVerse file that lists processes. After you invoke a menu, you can choose an option by entering a corresponding number. You can create menu records using the menu processor or the Editor.

The VOC entry that selects and displays a menu contains the name of the UniVerse file where the menu record is stored and the record ID of the menu.

A menu VOC entry contains the following:

Line	Field contents
001:	M [*description*]
002:	*filename*
003:	*record ID of the menu*
004:	Y

To display the menu records defined in the VOC file, use the `LISTM` command.

The `Y` *audit mode* option suppresses recording `DAT.BASIC.*` event entries during audit logging for commands that call BASIC programs. This is the default functionality. Note that this option cannot be used to bypass BASIC event logging in user-created VOC entries.

PQ: ProVerb (PROC)

A proc defines a sequence of operations to be performed by the ProVerb processor. ProVerb is used for performing minor programming tasks that use the resources of UniVerse. A proc is like a job control language (JCL) on other systems. A proc can perform all the operations of a stored paragraph and has additional capabilities, such as control looping.

Lines of a proc take arguments passed from the command line, user-prompted input, and stored tests to build UniVerse commands and submit those commands to the UniVerse processor for execution. Any UniVerse command can be executed from a proc.

A ProVerb VOC entry contains the following:

Line	Field contents
001:	PQ [N] [*description*]
002:	*proc statement*
003:–*nnn*:	[*additional proc statements*]

If the ProVerb entry calls another proc stored elsewhere, the command transferring control to the stored proc is usually put in a proc statement (in 002), but it can be put on any line of a proc. Control is not returned to the calling proc.

The format of the `proc` command that transfers control to the stored proc is as follows:

(*filename, record.ID*)

Both *filename* and *record.ID* must be enclosed in parentheses. Use the `LISTPQ` command to list the procs in the VOC file.

X: User record

A user record stores information defined by the user. Use the Editor or ReVise to create a VOC user entry. A VOC user entry contains the following:

Line	Field contents
001:	X [*description*]
002:	[*user information*]
.	
.	
.	
nnn:	[*optional information*]

X is the record type and description. Use the remaining fields to store user-defined information. Use the `LISTO` (list other) command to list the X entries in the VOC file. (`LISTO` also lists D and I entries contained in the VOC file.)

RELLEVEL and STACKWRITE records

RELLEVEL and STACKWRITE are two X records defined in standard VOC files.

RELLEVEL defines the release level of the software and verifies that the VOC file has been updated to the new level. If the UniVerse release level does not agree with that in RELLEVEL, UniVerse displays a warning message and requests permission to update the VOC file. Using an out-of-date VOC file can cause errors.

STACKWRITE tells the sentence stack processor whether or not to maintain the sentence stack after you log out. If the second field of this record contains ON, the sentence stack is saved when you log out of the session. If this field contains OFF, the sentence stack is not saved.

INTR.KEY, QUIT.KEY, and SUSP.KEY records

Standard VOC files contain three X records that identify the break options that are available when the Intr, Quit, or Susp key is pressed. These X records are called INTR.KEY, QUIT.KEY, and SUSP.KEY.

With these X records defined in the VOC file, when the Intr, Quit, or Susp key is pressed, a message like the following appears:

Break: Option (A, C, L, Q, ?) =

Each option performs a different action. You can remove any or all of these options if you do not want to make them available in the current UniVerse account. If only one option is specified, the action associated with that option occurs when a key is pressed, and no message appears.

Note: UniVerse loads the values for INTR.KEY, QUIT.KEY, and SUSP.KEY when UniVerse starts. Any modifications to the values do not take effect until you restart UniVerse.

Tailoring the VOC file

UniVerse contains an account called UV that is used for system administration. When a UniVerse account is created, the contents of a file called `NEWACC` in the UV account are copied to the VOC file in the newly created account. There is a different `NEWACC` data file for each UniVerse flavor. This ensures

that every new account begins with a correct standard set of verbs, sentences, paragraphs, file names, keywords, and menus.

You can tailor the VOC file of any account by adding, changing, or deleting records from the account's VOC file. Utilities such as `CREATE.FILE`, stack commands such as `.S`, and commands such as `SETFILE` implicitly add entries to the VOC file. The Editor, ReVise, or a BASIC program can also be used to add new VOC entries.

The system administrator can periodically refresh an account's VOC file by inserting updated entries from the `NEWACC` file into the VOC file.

Pointing to records in the VOCLIB file

The VOCLIB file is another standard UniVerse file that is created when a new account is created. It is empty when a new account is created. You should store your long paragraphs in the VOCLIB file and use remote command entries in the VOC file to point to records in VOCLIB. This practice keeps the VOC file from becoming too large, which can slow down performance.

How the VOC file is used

The VOC file is essential to the operation of every UniVerse processor. When you enter a command at the system prompt, the command processor takes the first word in the command as the record ID of an entry in the VOC file. If the first word is not defined in the VOC file as a verb, sentence, paragraph, proc, menu, or remote command entry, you receive an error message.

If the first word is defined in the VOC file as the proper type, its VOC entry tells the process that executes it what to do with the command line. For instance, if the word is a verb, the command processor checks the following:

- Field 2 of the VOC entry to determine the name of the program to execute
- Field 3 to determine whether the command is a BASIC program, operating system command or shell script, RetrieVe command, and so on
- Field 4 to determine what further setup is necessary before the command is executed—for example, whether the command uses a select list or whether the terminal mode should be changed

Listing selected VOC file entries

Throughout this section, several commands are mentioned that let you view the VOC file. The following table summarizes these commands.

Sentence	Description
`LISTF`	Lists all file definitions.
`LISTFL`	Lists definitions of files stored in this account.
`LISTFR`	Lists definitions of files stored in other accounts.
`LISTK`	Lists all keywords.
`LISTM`	Lists all menu selector records.
`LISTO`	Lists "other" entries.
`LISTPA`	Lists all paragraphs.

Sentence	Description
LISTPH	Lists all phrases.
LISTPQ	Lists all procs.
LISTR	Lists all remote command entries.
LISTS	Lists all stored sentences.
LISTSL	Lists all verbs that can use a select list.
LISTUN	Lists the operating system commands.
LISTV	Lists all verbs.

UniVerse sentence stack

The command processor stores a copy of any sentence or command line statement that you enter at the system prompt. By default the sentence stack preserves up to 99 sentences from your current session (the size of the sentence stack is configurable by the system administrator). Each sentence is numbered from 01 through 99. The most recently entered sentence is 01, the oldest is 99.

You can use sentence stack commands to list, save, edit, and delete sentences, and to recall, execute, and insert new sentences in the stack. The sentence stack commands are listed in the following table.

Command	Description
.A	Adds text to end of a sentence.
.C	Changes a sentence in the sentence stack.
.D	Deletes a sentence from the sentence stack or deletes a sentence or paragraph from the VOC file.
.I	Inserts a new sentence into the sentence stack.
.L	Lists the sentences in the sentence stack or displays an entry from the VOC file.
.R	Recalls a sentence in the sentence stack, or recalls a sentence or paragraph from the VOC file to the sentence stack.
.S	Saves a sentence or paragraph as an entry in the VOC file.
.U	Converts a sentence to uppercase.
.X	Executes a sentence in the sentence stack.
.?	Displays help about sentence stack commands.
?	Use at the end of a line to save an input line with a mistake. It does not execute and you can correct the error.

Saving the sentence stack

The STACKWRITE entry in the VOC file lets you save the sentence stack when you leave UniVerse, so that it is available the next time you use UniVerse. STACKWRITE is an X-descriptor that uses field 2 to indicate that the sentence stack retention should be turned on or off. Unless you set STACKWRITE to ON, the sentence stack is not saved when you leave UniVerse. The entry looks like this:

```
       STACKWRITE
 001: X-type.If field 2='OFF',then stack will not be maintained
after logoff.
 002:ON
```

When sentence stack retention is on, UniVerse saves the stack as a record in the &SAVEDLISTS& file. The record name has the form:

&&S.*username.line#*

username is the name you used to log on to the system. *line#* is the logical line number that identifies the user's point of entry (terminal port) to the system. Use the WHO command to display your *line#* and *username*.

Loading a saved or empty stack

Depending on how your system is configured, you may or may not be assigned the same line number each time you log on to UniVerse. If the same line number is assigned, the sentence stack that was saved when you last logged out is loaded as the current stack when you next log on to the system.

If a different line number is assigned, you get a new (empty) sentence stack, unless a saved stack exists in the &SAVEDLISTS& file with both of the following:

- The same *username* as your current login name
- The same *line#* as your current line number

In that case, the saved stack is loaded as your current stack.

Using the GET.STACK command

If you are assigned a different line number the next time you log on to the system, and if you want to use the sentence stack you were using during your last login session, use the GET.STACK command to load the saved stack from the &SAVEDLISTS& file into the current sentence stack.

Using the SAVE.STACK command

Using the SAVE.STACK command, you can save the current sentence stack to the &SAVEDLISTS& file under any name you like. Simply use the SAVE.STACK command followed by the name under which you want to save the stack. Use the GET.STACK command to retrieve any saved sentence stack and load it as the current sentence stack. The retrieved stack replaces the previously loaded stack.

Using paragraphs

A paragraph is a command block made up of one or more commands or sentences. Paragraphs are stored as records in the VOC file and are identified by the type code PA. To execute a paragraph, you enter its record ID (and any arguments the paragraph is designed to accept) at the system prompt, just like any other UniVerse command.

Usually a paragraph consists of a sequence of commands that you often execute together. Saving these commands as a paragraph lets you automate repetitive tasks. Here is an example:

```
    MAKE.LIST
001 PA
002 SELECT ORDERS WITH FILLED = "Y"
003 SAVE.LIST FILLED.ORDERS
```

This paragraph creates a select list of orders that have been filled and saves the list under the name `FILLED.ORDERS`.

A paragraph can do the following:

- Contain any command that can be entered at the UniVerse prompt
- Contain comments and blank lines
- Accept arguments, such as *filename*
- Prompt for input
- Control the execution sequence of commands

Commands in a paragraph are executed in sequence. When all of a paragraph's commands have been executed, control returns to the process from which the paragraph was invoked (system prompt, a sentence or another paragraph, a proc or UniVerse BASIC program, and so on).

Creating a paragraph

There are three ways to create a paragraph. You can create a paragraph using the following:

- ReVise
- The UniVerse Editor
- The sentence stack command `.S`

If a command is too long to fit on one line, you can continue it on successive lines. Use the continuation character _ (underscore) at the end of each line to be continued.

Warning: When you create a paragraph, be sure it does not inadvertently invoke itself. If it does, executing it will cause the paragraph to loop endlessly until you abort it with the Break key. This sort of thing most often occurs in a sequence of paragraphs in which one paragraph invokes another, and a paragraph later in the sequence invokes a paragraph that has been invoked earlier.

The following sections give detailed information about using paragraphs.

Using inline prompts in paragraphs

You can leave some values unspecified until the paragraph is invoked. The user supplies the missing value by responding to the prompt. This is the syntax for an in-line prompt in paragraphs:

```
<< [ control, ] text [ , option ] >>
```

control is an option that controls the display of the in-line prompt.

text is the text of the prompt.

option verifies that the prompt response matches a specific pattern. It can be an input (`ICONV`) conversion code or a matching pattern. Enclose conversion codes in parentheses.

The following example displays the prompt `Enter state` repeatedly until the user presses **Enter**. The option 2A specifies that the user must enter two alphabetic characters.

```
<<R,Enter state,2A>>
```

All prompts are forgotten within menus unless you use the `P` control option.

Controlling paragraph execution

Four paragraph commands control the execution of paragraphs:

- IF
- GO
- LOOP
- DATA

The IF command

The `IF` command introduces a conditional statement and changes the execution of the paragraph based on the result of an expression. The syntax is as follows:

`IF` *`expression`* **`THEN`** *`statements`*

The syntax of *expression* is as follows:

`value1 operator value2`

value1 and *value2* can be constants, inline prompts, or *@variables*. A constant can be a number or a string. If the value includes spaces, enclose it in single or double quotation marks.

For a list of relational operators, see *UniVerse BASIC*.

You can use the following *@variables* with the `IF` command:

- @DATE
- @DAY
- @LOGNAME
- @MONTH
- @SYSTEM.RETURN.CODE
- @TIME
- @USERNO
- @USER.RETURN.CODE
- @WHO
- @YEAR

The GO command

The `GO` command transfers control to the statement specified by a label.

Note: Control can only be transferred in a forward direction. If you require both forward and backward directions, you should use a proc rather than the `GO` command.

The syntax is as follows:

`GO` *`label`*

label identifies a labeled command within the paragraph. The label can be a number from 0 through 9 or a name. The label must be followed by a colon and a space. `NEXT` and `MAIN` are labels in this example:

```
002: IF <<Want a report?>> = "N" THEN GO NEXT
```

```
003: LIST CUSTOMERS
004: NEXT: IF <<Want the Account Menu?>> = "N" THEN GO MAIN
005: INVOKE.ACCOUNT.MENU
006: MAIN: INVOKE.MAIN.MENU
```

The LOOP command

The LOOP command begins a program loop. The LOOP repeats until a condition matches the condition defined in a loop statement. The syntax is as follows:

```
LOOP
     statements
     termination statement
REPEAT
```

Use *termination statement* anywhere between LOOP and REPEAT to provide the exit from the loop.

You must use the A (always prompt) control option with an inline prompt in a loop. The following example of a loop asks you for reports to print until you answer N to the in-line prompt:

```
002: LOOP
003: IF <<A,Do you want a report?>> = "N" THEN GO FINISHED
004: LIST <<A,Enter a filename >> LPTR
005: DISPLAY Your report has been spooled.
006: SPOOL -LIST
007: REPEAT
008: FINISHED: DISPLAY You are now at the UniVerse prompt.
```

The DATA command

Use the DATA command to put values in an input stack to serve as responses to input requests from UniVerse BASIC programs or from verbs called by a paragraph. DATA commands can be used only in a paragraph. The syntax is as follows:

DATA *data*

When the program requires data that would normally be entered at the keyboard, the *data* of the first DATA command is used. The second request for data is answered by the second DATA command, and so on.

You can use an inline prompt as *data*, but you cannot use the DATA command to respond to an inline prompt.

Here is an example:

```
002: RUN DUE.LIST
003: DATA <<Enter date>>
004: DATA Yes
005: RUN VENDORS
006: DATA <<Enter month>>
```

These DATA commands respond to requests for input in the DUE.LIST and VENDORS programs.

Chapter 4: Creating a database

This chapter describes how a standard UniVerse database is organized. It also explains how to create a database and how to design, create, and build files.

For information describing how a UniVerse SQL database is organized, how to create schemas and tables, and how to convert a UniVerse account to a schema, see *UniVerse SQL Administration for DBAs*.

Database structure

A UniVerse database comprises one or more UniVerse files. A UniVerse file is a related group of one or more data files and a file dictionary. The relationship among these files is defined by an entry in the VOC file. The record ID of the VOC file entry is the name of the UniVerse file.

UniVerse data files and their dictionaries are implemented using system files and directories. The paths of the data files and the file dictionary are contained in the VOC file entry.

Defining UniVerse files

Assume that you use the `CREATE.FILE` command to create a UniVerse file. For example, the following command creates a UniVerse file definition called PAYROLL, the data file, and the file dictionary:

```
>CREATE.FILE PAYROLL
```

For the data file, `CREATE.FILE` creates a system file named PAYROLL in the directory where the UniVerse account is located. For the dictionary, `CREATE.FILE` creates a system file named D_PAYROLL.

You can use the UniVerse Editor, ReVise, or a UniVerse BASIC program to define entries in the file dictionary. These entries determine the structure and function of fields in the data file.

An example of the information and structure that might be stored in the PAYROLL file is shown in the following table.

Order	Item	Description
1	Record ID	Employee's Social Security number
2	Field 1	Last name
3	Field 2	First name
4	Field 3	Street address, possibly containing multiple lines
5	Field 4	City
6	Field 5	State
7	Field 6	Zip code
8	Field 7	Salary history, possibly containing multiple *salary* values, which in turn contain the subvalues *salary amount* and *effective date*

The PAYROLL file might be set up in an account accessible only by members of the accounting department.

In other accounts that are used by people in other departments, you can create another UniVerse file definition called EMPLOYEES that allows access only to the name and address fields of the PAYROLL file. A VOC file entry for the EMPLOYEES file might contain the data file name PAYROLL and a dictionary

file name D_EMPLOYEES. The EMPLOYEES dictionary would not have an entry describing the field that contains salary history.

The relationships among dictionaries and data files in a database can form a complex network that allows different departments in an organization to share data.

Creating the data file and Its dictionary

Usually both the data file and its dictionary are created at the same time. When creating a single data file and its dictionary, UniVerse uses only fields 1, 2, and 3 in the VOC F-descriptor entry. For example, the VOC entry might look like this:

```
    ACCT.REC
001 F
002 ACCT.REC
003 D_ACCT.REC
```

You can create a file dictionary without creating a data file. You might do this if you are using an existing data file and want to create a new dictionary for it. When you create a file dictionary without creating a data file, only field 3 in the VOC file F-descriptor contains a path. For example, the entry might look like this:

```
    EMPLOYEES
001 F
002
003 D_EMPLOYEES
```

You can create a data file without creating a dictionary. You might do this if you want to use an existing dictionary with a new data file. When you create a data file without creating a dictionary, only field 2 in the VOC F-descriptor contains a path. For example, the VOC item might look like this:

```
    SALES.CLOSE
001 F
002 CLOSE
003
```

Use the Editor or ReVise to add the name of the existing file to the VOC entry.

A dictionary can also contain pointers to data in another data file. Thus the dictionary for the `ACCT.REC` file could have a pointer to the SALES.CLOSE file to get data, such as the amount of a sale.

Creating the database

To create the most usable database, it is important to perform these steps:

1. Design the files that will make up your database.
2. Decide what is the best file type for each of your files.
3. Estimate the modulo and separation for each hashed file.
4. Use `CREATE.FILE` to allocate space for the files and to create the VOC file entries.
5. Use ReVise or an editor to build the contents of the file dictionary.
6. Use ReVise, an editor, or UniVerse BASIC programs to add data to the data file.

Designing the files

The first step in creating a database is to design the files. File design begins with general information and becomes more specific. You must determine what elements will constitute your database.

When designing files, keep in mind that one of the most important features of UniVerse is the ability to make changes to the database as it grows and as its use changes. Once files have been defined you can at any time modify file definitions and redefine relationships between files.

Follow these steps to design your database:

1. Decide the following:
   - What data you need to organize in the file
   - How to organize the data
   - How you will use the data
2. Decide how many files you will need.
3. Decide what the file type should be: hashed (static or dynamic), B-tree, or type 1 or type 19 (nonhashed).
4. Design each hashed file or B-tree file as follows:
   - Organize the information into fields and name the fields. Use field names that are short and easy to remember, such as LNAME for last name and FNAME for first name.
   - Decide what the record ID for each record will be. The record ID must be unique for each record. For example, in a payroll file, the record ID might be the Social Security number. The character of the record ID determines the file type.
   - Estimate the average number of characters each field will contain.
   - Estimate the number of records that the file will contain.
   - Decide on the format to be used to display the data in each field in a report.
   - Decide whether to create secondary indexes on any fields.

Guidelines for steps 1 and 2 are beyond the scope of this manual. If you are designing a database for the first time or need more in-depth information, see any good book about database design. The sections that follow discuss considerations for steps 3 and 4.

File design objectives

When you create a UniVerse file, you use a file type suited to the data it will contain. You can choose a static hashed file (types 2 through 18), a dynamic file (type 30), a B-tree file (type 25), or type 1 or type 19. The type of static hashed file you choose depends on the characteristics of the record IDs for the file.

Each record in the file must have a unique record ID. Choosing records by record ID is the fastest way to retrieve records. Social Security numbers, invoice numbers, telephone numbers, part numbers, or product names are examples of information often used as record IDs.

Choosing records by values in a field takes much longer than choosing records by record ID, unless the field has a secondary index.

Guidelines for choosing file type 1 or 19 (nonhashed)

Choose a file type of 1 or 19 if your file contains the following:

- A relatively small number of records that are very large and that are accessed infrequently

- Program source code
- Text
- Data that is to be processed with standard operating system processes and utilities
- Records that cannot be structured into fields and that are accessed infrequently

When creating type 1 or type 19 files, you need to be sure that the file fits the description of a nonhashed file. Keep in mind that type 1 and 19 files can become unwieldy if they contain a lot of records. Some systems impose a limit on the number of records that can be stored in such files.

Guidelines for choosing other file types

If your file contains any kind of data other than the items bulleted in the preceding section, choose one of the hashed file types or a B-tree file. If your file will not grow or shrink dramatically over short periods of time, choose a static hashed file type. If the number of records in the file will change much and frequently, consider a dynamic file.

For static hashed files, choose one of the file types from 2 through 18, basing your choice on the characteristics of the record ID. Each of the 17 static hashed file types has an appropriate hashing algorithm for the kind of record ID values in the file, so that records get distributed as evenly as possible. For dynamic files, choose file type 30. For B-tree files, choose file type 25.

Allocating space for the files

It is important to size static hashed files properly so that data can be accessed quickly and efficiently. The modulo and separation specify the amount of space to be allocated for the file. They are based on the amount of data to be stored in the file, the size of records in the file, and the expected growth of the file.

If a static hashed file grows too large for its space, retrieval time is much slower. On the other hand, a modulo and separation that are larger than necessary waste the unused disk space. Modulo and separation do not apply to a nonhashed file.

Effect of modulo and separation on UniVerse performance

Your choice of file type, modulo, and separation can have a dramatic effect on UniVerse's performance in accessing a file. The modulo and separation of a static hashed file tell UniVerse three things:

- How many groups are in the file
- How large the groups in the initial allocation are
- What size blocks get added to a group when the records hashed to it outgrow the initial allocation

Specify type, modulo, and separation to ensure the following:

- Each group has only a few records in it. (The more records there are in a group, the longer the search operation.)
- There are no group overflows.

The modulo determines the number of groups used to store the records. The separation specifies the size of the group buffers in 512-byte units.

Creating files

When you complete your file design, create file structures using the `CREATE.FILE` command. `CREATE.FILE` does the following:

- Specifies the file type.
- Allocates space for the data file and its dictionary.
- Creates a VOC file entry that defines the UniVerse file.
- Creates a default data descriptor called *@ID* in the dictionary. This descriptor defines the record ID field for records in the data file.

The syntax of `CREATE.FILE` varies, depending on the type of file you are creating, Any or all of the parameters for the verb `CREATE.FILE` can be specified in one command.

CREATE.FILE prompts for parameters

The following simple example of the `CREATE.FILE` command creates a static hashed file called `PAYROLL` as type 2 with a modulo of 7 and a separation of 1:

```
>CREATE.FILE PAYROLL 2 7 1
```

Instead of specifying all the parameters on the command line, you can enter just `CREATE.FILE` at the system prompt. `CREATE.FILE` prompts you for the required file parameters. The following sequence of prompts appears:

```
File name =
```

Enter a name that identifies the UniVerse file. To create only a data file or dictionary, enter `DATA filename` or `DICT filename`. If you do not use the `DATA` or `DICT` keyword, UniVerse creates both the dictionary and the data file. The file name of the data file is *filename*, and the file name of the dictionary is D_*filename*.

```
File type =
```

Enter a numeric value that is the file type, or enter **?** to display a table of file types. If you specify a file type of **1, 19, 25, or 30**, there are no prompts for modulo and separation. For a static hashed file, a type of 2 through 18 should be chosen based on the characteristics of record ID values.

```
Modulo =
```

Enter the number of groups that are needed to store records in the file. The modulo should be sufficiently large to ensure a good distribution of records in each group. It should also be a prime number to ensure even record distribution.

UniVerse does not display the following two prompts for a static hashed file:

```
Separation =
```

Enter the number of 512-byte segments needed for each group buffer. A separation of 1, 2, or 4 is recommended to ensure efficiency so that groups are not split across disk blocks (the file layout is system-dependent). If the expected record size is unknown, use a separation of `4`.

```
File description =
```

You can optionally enter a brief description of the file. The description is part of the file definition record in the VOC file and is displayed as a Help response when you enter `? filename` at the system prompt. When you create a static hashed file, if you include only *filename*, *type*, and *modulo* on the command line, you are not prompted to provide a description.

Modifying a VOC file definition

You can use the Editor or ReVise to modify any file definition entry in the VOC file. This is usually done when you associate a file dictionary with more than one data file, or when you associate a data file with more than one file dictionary.

You can use the `RESIZE` command to modify any file's type, modulo, or separation. The following sections tell how to determine the correct name, type, modulo, and separation for a file.

Naming files

File names in UniVerse can contain any character except `CHAR(0)`, including spaces and control characters. However, if you use spaces and control characters, enclose the file name in quotation marks to tell UniVerse that they are part of the name. If the file name contains only alphabetic, numeric, and punctuation characters, quotation marks are not needed to delimit the string. For example, enter the following to name a file SALES.TRACKING:

```
>CREATE.FILE SALES.TRACKING
```

Spaces and control characters in filenames

If you use spaces and control characters in your file names, you must enclose the entire file name in quotation marks. For example:

```
>LIST "SALES TODAY" WITH AMOUNT GE 2000
```

In naming the operating system files and directories that contain the dictionary and data file, UniVerse changes characters that are not allowed in a path as follows:

On UNIX systems:

This character...	Maps to...
/	?\
?	??
empty file name	?0
. (leading period)	?.

On Windows platforms:

This character...	Maps to...
/	%S
?	%Q
empty file name	%
"	%D
%	%%
*	%A
:	%C
<	%L
|(vertical bar)	%V
>	%G
\	%B

This character...	Maps to...
↑ (up-arrow)	↑ up-arrow
ASCII 1 through ASCII 26	↑A through ↑Z
ASCII 27 through ASCII 31	↑1 through ↑5

Long file names

You can enter no more than 255 characters as a UniVerse file name. However, there may be limitations on the length of paths at your site.

On UNIX systems that allow only 14-character file names, UniVerse truncates a UNIX path to 9 characters and adds a 3-digit sequencer.

On Windows platforms, UniVerse is designed to run in the NT file system (NTFS). But Windows platforms also support the MS-DOS FAT file system, which limits file names to 8 characters with a 3-character extension and has a further set of characters that are not permitted in file names. UniVerse makes no special provision for these file names or special characters. If you want to store UniVerse files in an MS-DOS FAT file system, you must follow the MS-DOS conventions when you name UniVerse files.

On other systems the length of the file names is system-dependent. The UniVerse file name is not affected; only the paths are transformations of the UniVerse file name.

Note: `CREATE.FILE` fails with file names greater than 255 characters. You see the following error message:

```
Attempted WRITE with record ID larger than 255 characters.
*** Processing cannot continue. ***
```

Long record IDs

Record IDs cannot exceed 255 bytes. This means that the maximum number of characters in a record ID depends on the character set being used. For multibyte character sets, the safe limit is 85 characters, which allows each character to be three bytes long in the internal character set. This limit also applies to values used as keys in secondary indexes.

Long record IDs in a type 1 file

Long record IDs in a type 1 file are divided into 14-character segments, and UNIX subdirectories are created. If you specify a record ID longer than 14 characters, UniVerse creates a directory whose name consists of the first 14 characters of the record ID, then creates a file in that directory whose name consists of the next 14 characters of the record ID.

In the following example, BP is a type 1 file. If you use the Editor to create two records they look like this:

```
>ED BP THIS.IS.A.LONG.RECORDNAME
```

```
>ED BP THIS.IS.A.LONGER.RECORDNAME
```

The paths for these records look like this:

```
BP/THIS.IS.A.LONG/?.RECORDNAME
```

```
BP/THIS.IS.A.LONG/ER.RECORDNAME
```

In this case, `THIS.IS.A.LONG` is the name of a subdirectory in the BP directory. The names `?.RECORDNAME` and `ER.RECORDNAME` are two files in the directory `THIS.IS.A.LONG`. (If the

remaining record ID is longer than 14 characters, UniVerse creates another subdirectory under the first.)

Note: If you create a type 1 file and then use operating system processors (such as the UNIX vi editor) to create records in this file, you may inadvertently create records that are not accessible to UniVerse processors. For example, if you create a file, BP, as a type 1 file and issue the following command, you create a record with the name `MORETHAN14CHARS`. This record is inaccessible to UniVerse processors like `BASIC, RUN, or ED`.

```
>VI BP/MORETHAN14CHARS
```

Long record IDs in a type 19 file

UniVerse does not truncate the names of records in a type 19 file. On systems that support file names longer than 14 characters, you can create longer paths for records in type 19 files.

Note: On systems that have a 14-character filename limit, UNIX calls will truncate at 14 characters.

File types

There are 21 different file types. A file type is specified by numbers 1 through 19, 25, or 30.

- Types 1 and 19 are nonhashed file types used to contain UniVerse BASIC programs and other data that is organized into records that are loosely structured.
- Types 2 through 18 are static hashed UniVerse files. The different hashing algorithms are designed to distribute records evenly among the groups of a file based on characters and their positions in the record IDs.
- Type 25 is a B-tree file.
- Type 30 is a dynamic hashed file.

Nonhashed file (type 1 and type 19)

When you create a type 1 or type 19 file using `CREATE.FILE`, UniVerse creates a hashed file dictionary and a directory for the nonhashed data file. As you create records in the nonhashed file, the files for the records are created and filled with the data you enter.

Static hashed file (types 2–18)

When you create a static hashed file using `CREATE.FILE`, UniVerse creates a hashed file dictionary and a file for the hashed data file. The file size is determined by the modulo and separation you specify with `CREATE.FILE`.

B-Tree file (type 25)

In a B-tree file, records are stored in order, sorted by record ID.

A search for a record in a B-tree file starts by comparing the value of the record's ID with the value at the center of the tree. If it is greater, the search continues with the subtree to the left of the center. If it is less, the search continues with the subtree to the right of the center. This decreases the number of records that must be searched to find a record.

You should not store large records in type 25 files—records larger than 1K increase access time significantly.

Dynamic file (type 30)

The dynamic file automatically resizes itself. If the number of records in a static hashed file changes, you must change its modulo and separation to continue operating efficiently. For files that change size frequently, this can be problematic.

Dynamic files increase file size automatically when the data in the file exceeds a specified amount, called the split load. The file size decreases automatically when the data in the file is less than a specified amount, called the merge load. In this way dynamic files are sized efficiently even when the amount of data in them changes often.

The space released when a dynamic file shrinks is not given back to the file system; it remains part of the file and is used the next time the file grows.

You can use a dynamic file just as you would use a static hashed file.

`CREATE.FILE` creates a dynamic file with a group size of 1, or 2048 bytes, which is equivalent to a separation of 4. It uses a general hashing algorithm and has an initial modulo of 1.

Distributed file

A distributed file is a composite hashed UniVerse file made up of a set of existing UniVerse files. Each file in this set is called a part file of the distributed file.

Each part file must exist before a distributed file can be defined. A part number must be associated with each part file when a distributed file is defined using the `DEFINE.DF` command.

When you use the `DEFINE.DF` command, UniVerse uses a partitioning algorithm to determine in which part file a record belongs. The partitioning algorithm must also be specified when the distributed file is defined. You use distributed files when you want to logically organize data files into a hierarchical arrangement by functional groups.

Modulo and separation

The modulo and separation that you specify with the `CREATE.FILE` command allocate the disk space for a hashed file.

Modulo

The modulo of the file directly influences on how evenly records are distributed among the groups of the file. The best performance is achieved when records are distributed evenly among groups.

The modulo should be large enough so that the expected number of records in a group fits into the buffer allocated for the group. The expected number of records in a group is the total number of records divided by the modulo.

To minimize hashing conflicts and ensure even distribution of records, the modulo should be a prime number. Use the `PRIME` command to generate a prime number close to the modulo you estimate.

The average number of records stored in each group is known as the group depth. The expected group depth for a file can be calculated by dividing the total number of records you expect to store in the file by the number of groups in the file (the modulo). For example, if you expect to have 808 records in a file and the modulo is 101, the expected group depth for that file is 8 (808 divided by 101).

Choose a modulo large enough so that the expected group depth times the average record size is less than the size of the group buffer (specified by the separation). In other words, be sure that the modulo is large enough so that all the records hashed to a group fit into the initial allocation for that group.

To determine in which group a record is stored, a hashing algorithm is applied to the record ID. The result from this hashing algorithm is divided by the number of groups, and the remainder is the number of the group to which the record hashes.

One way to express this is as follows:

hr/m = group#

hr is the result from the hashing algorithm applied to the record ID value. *m* is the number of groups in the file.

As an example, if a file has a modulo of 7 (there are seven groups in the file) and the result from applying the hashing algorithm to a record ID is 47, the group number for that record is 5:

47/7 = 5

As you place records in the file, the groups in which they are stored fill up. When that happens, UniVerse allocates additional blocks of storage to the groups that are full. The size of the additional blocks is specified by the separation.

Separation

Separation specifies the physical size of a group buffer in 512-byte units. The most commonly used separations are 1, 2, and 4. A separation of 1 specifies a group buffer size of 512 bytes. Separations of 2 and 4 specify group buffers of 1024 and 2048 bytes, respectively.

It is recommended that you not specify an odd number (except 1) so that groups are not spread across disk blocks. The file layout is system-dependent.

Important effects of separation on a file

The separation of a file has two important effects. The first is on how efficiently the space allocated to a file is used. You should choose a separation in which the group buffer size is slightly larger than a multiple of the average record size plus UniVerse overhead of 12 bytes.

For example, if the average space that a record uses is 202 bytes, a separation of 1 would be a poor choice, because only two records would fit into a 512-byte group buffer, leaving 108 bytes of each group buffer unused. A separation of 2 would be better since five records would fit into a 1024-byte group and only 14 bytes would be unused.

The second effect that separation has on the file is that it determines how many overflow group buffers must be added to the initial space allocation. In the previous example, a separation of 2 would minimize wasted space if the expected group depth were 5. If the expected group depth were 8, however, a separation of 2 would not be large enough to hold the expected data. When a sixth record is written to the group, a 1024-byte overflow buffer is allocated to the group.

Overflow group buffers greatly increase the amount of disk activity needed to process them. Increasing a file's separation does not always solve the problem, because on many systems a large group buffer size also increases the amount of disk activity needed to process the file. In the example, the best solution is to increase the modulo to keep the group depth low and to eliminate the need for overflow group buffers.

Summary of file sizing

In summary, the modulo and separation of a file should keep the group depth low (few records per group), use disk space efficiently (little wasted space in each group buffer), and avoid having overflow

blocks. It is a good idea to use a separation of 1, 2, or 4, and to vary the modulo so that no overflow blocks are needed.

Always consider the effect modulo and separation have on each other. Consider the expected number of records in a group when determining the separation. If the group buffer is not large enough, overflow buffers must be used. Overflow buffers require additional physical disk I/O and slow down file access.

If you cannot estimate the average record size, begin with a separation of 4 and resize the file as it becomes full. To maintain an efficient file structure, use the file maintenance commands:

ANALYZE.FILE	HASH.HELP
CONFIGURE.FILE	HASH.HELP.DETAIL
FILE.STAT	HASH.TEST
FILE.USAGE	HASH.TEST.DETAIL
GROUP.STAT	RECORD
GROUP.STAT.DETAIL	RESIZE
HASH.AID	

Building the file dictionary

After you create the data file and its dictionary, you can build the contents of the dictionary. Use ReVise, the Editor, or a UniVerse BASIC program to build the dictionary.

ReVise displays a sequence of input prompts for defining dictionary records. These prompts make ReVise easy to use, especially when you first begin using UniVerse.

The UniVerse Editor is a line-oriented editor that does not display field names as prompts for input. When you first begin using UniVerse and are not familiar with the fields in a file dictionary, the Editor can be difficult to use to build a dictionary. However, once you are familiar with the structure of file dictionaries, you may find that you prefer to use the Editor.

Adding data to the data file

After you have defined the contents of the file dictionary, you can add data to the data file. Use ReVise, one of the editors, or a UniVerse BASIC program to add data to a data file.

When you are learning to use UniVerse, you may find it easier to use ReVise to add data, because ReVise displays the field names as prompts. Another helpful feature of ReVise is that it converts input to a special internal storage format when required. For example, dates that you enter in a format such as 07/18/91 are converted to a format like 8600, the format UniVerse uses to store the date internally.

To use ReVise for adding data, use the `REVISE` command:

`REVISE` *`filename`* `[` *`fields`* `] … ]`

filename is the file to which you are adding data. *fields* is the name of one or more fields to which you want to add data.

Since the Editor does not display field name prompts or perform input conversions, you need to be familiar with the fields in the data file before you use the Editor to add data.

Secondary indexes

A secondary index is a sorted list of the values in a specified field. Secondary indexes work like B-tree files in that they allow fields other than the record ID to be used as the key field in selection and sort expressions. Take as an example the following selection sentence:

```
>SELECT FILE WITH F2 < 100
```

Without secondary indexing, every record in the file named FILE is examined for records with the value of field 2 less than 100 to create a select list. If field 2 has a secondary index, however, each item in the index is examined in increasing order until an item is found that is greater than 100. All other items are assumed not to match and the search is ended.

If all the items in the file have a value less than 100 in field 2, the search time is the same as without a secondary index, but if only a few items have a value less than 100, the search time is greatly reduced.

Creating and building secondary indexes

Setting up secondary indexes is a two-step process. First you create the indexes, then you build them. When you create secondary indexes for a file, other users can read or write to the file. However, when you build secondary indexes, others cannot use the file, unless you specify the `CONCURRENT` keyword.

> **Note:** If you are building an index for a field in the data file, write permissions are required on the dictionary of the file whose indexes are being built. If you are building an index for a field in the file dictionary, write permissions are required on the system file `DICT.DICT` in the UV account.

Once you create and build an index, it contains a sorted list of the values in the indexed field—the keys of the index—along with the IDs of the records in the indexed file. If the indexed field is multivalued, each value is indexed, even if the field contains duplicate values in the same record.

Each unique value in the indexed field is stored as the ID of a record in the secondary index. Each secondary index record comprises one or more fields containing the keys to the records in the indexed file.

For example, the indexed file can contain the following fields:

```
RECORD IDS...           LNAME......

 111-888-3333           SMITH
 222-555-6666           JONES
 888-444-9999           SMITH
```

A secondary index on the field `LNAME` displays those fields differently:

```
RECORD IDS...       FIELD 1.....................
JONES               222-555-6666
SMITH               111-888-3333 F888-444-9999
```

The F represents a field mark.

When an index is first created, it contains no values. As new values are added to the indexed file, corresponding values are also added to the index. If the indexed file contains data at the time the index is created, data in the file is not included in the index, and `CREATE.INDEX` displays a warning that the index needs to be built. Use the `BUILD.INDEX` command to include existing file data in a newly created index.

Creating secondary indexes

Use the CREATE.INDEX command to create secondary indexes for a file. The CREATE.INDEX command creates the indexes, one for each field specified. The simplified syntax of the CREATE.INDEX command is as follows:

```
CREATE.INDEX [ filename ] [ fields ] [ NO.NULLS ]
```

When an index is created for a field defined by a D-, A-, or S-descriptor, any other D-, A-, or S-descriptor that references the same field in the indexed file also uses the secondary index unless one descriptor defines the field as multivalued and the other descriptor defines the field as singlevalued.

NO.NULLS is the keyword that specifies that empty secondary key values will not be indexed in the newly created indexes. Disk space and processing time can be saved by using this facility.

Note: Reports generated using empty-string-suppressed secondary keys will not contain values from records which would otherwise be referenced by the empty keys.

To be indexed as multivalued, a field must be defined in the dictionary as a multivalued field. Singlevalued fields are indexed as single keys without regard to value marks or subvalue marks.

Using another file dictionary

If the field you want to index is defined in a dictionary other than that of the specified file, use the USING clause to specify the other dictionary. This clause tells CREATE.INDEX where the fields are defined:

```
USING [ DICT ] dictname
```

The USING clause lets you use the dictionary of any file to create the index.

This example creates indexes for the file FILEA using the two fields FIELD1 and FIELD2 that are defined in the dictionary of FILEB:

```
>CREATE.INDEX FILEA FIELD1 FIELD2 USING DICT FILEB
```

Using another account

You can specify another UniVerse account to store an index for a file by specifying an AT clause. This lets you keep all your indexes in one directory.

```
AT account
```

The following example creates an index for FILEA using the field FIELD1 which is defined in the dictionary of FILEB in the account UV.JOHN:

```
>CREATE.INDEX FILEA FIELD1 USING DICT FILEB AT UV.JOHN
```

The next two commands create and list an index for the field FIELD1 for the file TEST in the SALES account:

```
>CREATE.INDEX TEST FIELD1 AT SALES
>LIST.INDEX TEST ALL
Alternate Key Index Summary for file TEST
File.......... TEST
Indices........ 1 (0 A-type, 0 C-type, 1 D-type, 0 I-type, 0 S-
type)
Index Updates.. Enabled, No updates pending
Index name  Type   Build       Nulls In DICT S/M    Just    Unique
Fld No
FIELD1      D      Not Reqd    Yes   No     S    L    N   1
```

If indexes already exist for a file, UniVerse displays a warning:

```
>CREATE.INDEX TEST FIELD3 AT UV.JOHN
Indices exist at /u1/SALES/I_TEST! AT clause ignored!
```

You cannot specify the AT clause for another set of indexes because all indexes are stored as files in one directory.

Building secondary indexes

When you create an index for a file that contains data, you need to build the index to bring it up to date with the data in the file. If you don't build the index, it will contain only index entries for data entered after it was created. The simplified syntax of the BUILD.INDEX command is as follows:

```
BUILD.INDEX [ filename ] [ indexes | ALL ][CONCURRENT] [RESET]
```

Deleting secondary indexes

Use the DELETE.INDEX command to delete a file's secondary indexes. The simplified syntax of the DELETE.INDEX command is as follows:

```
DELETE.INDEX [filename ] [ indexes | ALL ]
```

Enabling and disabling secondary indexes

When you create an index, it is automatically updated when new data is added to the file. When a lot of data is added to a file, updating the index can slow down the entry process and place an inordinate demand on system resources.

It is sometimes convenient to disable the automatic updating of index entries if a lot of data is to be added. The index can be updated later, when system use is low and data entry is complete. While automatic updating is disabled, however, reports generated using the index may be inaccurate.

Use the ENABLE.INDEX command to reenable automatic updating, and use the DISABLE.INDEX command to disable automatic updating. These commands use the following syntax:

ENABLE.INDEX [*filenames*]

DISABLE.INDEX [*filenames*]

Automatic updating is initially enabled. After disabling it, you can reenable automatic updating with the ENABLE.INDEX command. Note that you must separate multiple *filenames* by a blank space.

Enabling automatic updating does not actually update the index entries. Use the UPDATE.INDEX command for this. You can use UPDATE.INDEX when automatic updating is enabled or when it is disabled. The simplified syntax of the UPDATE.INDEX command is as follows:

UPDATE.INDEX [*filename*]

Once automatic updating is disabled, you must follow these steps to ensure that the indexes are completely up-to-date:

1. Reenable automatic updating, using ENABLE.INDEX.
2. Close all opened versions of the file system-wide, even if they are reopened immediately. This step ensures that no users are still operating with automatic updating disabled.
3. Use UPDATE.INDEX to bring the indexes up to date.

Listing secondary indexes

The LIST.INDEX command displays information about a file's secondary indexes. The simplified syntax is as follows:

LIST.INDEX [*filename*] [*indexes* | **ALL**]

Specify the indexes you want information about by specifying *filename* and the name of the indexed fields in that file, or the ALL keyword for all of the indexes.

LIST.INDEX displays a report containing general information about the specified secondary indexes. The top of the report displays the following general information about each index: the file name, the number and type of indexes present, the update mode of the index (enabled or disabled), and whether the index requires updating.

A detail line for each index provides the following information:

Item	Description
Index name	The name of the index.
Type	Either D, A, or S for a data descriptor, I for an interpretive descriptor, or C (A- or S-descriptor with correlative in field 8).
Build	Specifies whether the index needs to be built (Required) or not (Not Reqd).
Nulls	Specifies whether empty strings are indexed.
In DICT	Specifies whether the file dictionary contains a field definition with a record ID corresponding to the name of the index.
S/M	Either S for a single-valued index or M for a multivalued index.
Just	Either L or R from field 5 of the dictionary.
Unique	Specifies whether the field value is unique (Y) when the SQL CREATE TABLE statement is used or not (N).
Field number/ I-type expression	Specifies the field number if type is D, A, or S; the I-type expression if type is I; the correlative code from field 8 if type is C.

Using secondary indexes with distributed files

You can create and build indexes to use with distributed files in two ways. You can create and build indexes on individual part files or on the distributed file. In either case, UniVerse stores the indexes with each part file.

Accessing indexes on part files

You can create and build secondary indexes on the part files. When you access the indexes built on the individual part files, UniVerse treats these like any standard UniVerse file.

For more information, see Secondary indexes, on page 57.

Accessing indexes using distributed files

You can also access indexes using the distributed file. UniVerse uses the indexes defined on individual part files of a distributed file when you access the file as a whole if the indexes are common to all part files.

An index is common if you define the index on all part files belonging to the distributed file and these dictionary fields for the index are identical in each part file:

- Name of the index
- Type
- Singlevalued or multivalued specification
- Justification in format specification
- For D-descriptors, the field number
- For I-descriptors, the object code

Using indexed part files

Suppose that you have created part files that are associated with a distributed file from two standard, indexed UniVerse files. When you access the distributed file as a whole, UniVerse will only use the indexes that are common to all part files.

Using nonindexed part files

Assuming that the part files are not yet indexed, you can tell `CREATE.INDEX` and `BUILD.INDEX` to use the entries in the dictionary of the distributed file for indexing. An index is created for these entries for each part file. When UniVerse uses the dictionary of the distributed file, the indexes in each part file automatically correspond. File maintenance is easier since you do not need to maintain indexes for each part file individually.

The following example creates indexes using the two fields FIELD1 and FIELD2 that are defined in the dictionary of `DFFILE` as D-descriptors. `CREATE.INDEX` creates the indexes for each part file that is associated with `DFFILE` in the directory where the part file physically exists.

```
>CREATE.INDEX DFFILE FIELD1 FIELD2
```

You can also index I-descriptors in multiple part files. If you are using an I-descriptor defined in a local file, the compiled object code, the S/M specification, and the justification of the I-descriptor must be identical. If the I-descriptor is defined in a remote file, UniVerse does not check the compiled object code. The contents of field 2, the S/M specification, and the justification must be identical.

`CREATE.INDEX`, `BUILD.INDEX`, and `DELETE.INDEX` work for only those indexes whose fields correspond and whose names are identical.

If the dictionary entries in each part file correspond, you can report on these indexes by specifying the distributed filename. `LIST.INDEX` works for the indexes whose fields correspond; however, the names of the indexes may differ. (The index name in the report is that of the first part file you define with `DEFINE.DF`).

`LIST.INDEX` on the distributed file reports on the distributed file as the logical mechanism to access the information about the UniVerse common indexes. (Use `LIST.INDEX` on a part file to report indexes for that individual part file.)

Examples illustrating corresponding dictionary entries

Suppose you want to create and access secondary indexes using the distributed file `DIST.FILE`. For example, `DIST.FILE` has two part files, `PART1` and `PART2`. `PART1` has two fields, FIELD1 and FIELD2. PART2 also has two fields, FIELD1 and FIELD2. FIELD1 in `PART1` and FIELD1 in `PART2` have corresponding, identical dictionary fields for the following: type, S/M specification, justification for format, and field number.

The following examples create and build secondary indexes for FIELD1 in part files `PART1` and `PART2`. You can name only one part file to build the index for both part files (`USING` clause) because the field names are identical in both part files, and the fields correspond.

```
>CREATE.INDEX DIST.FILE FIELD1 USING DICT PART1
>BUILD.INDEX DIST.FILE FIELD1 USING DICT PART1
```

These examples show the correspondence of the dictionary entries for FIELD1 in part file PART1 with the entries for FIELD1 in part file PART2:

```
>LIST DICT PART1
DICT PART1       11:08:38am 05 Dec 1994 Page 1
Type &
Field..... Field. Field........  Conversion..  Column....... Output
Depth &
Name.....   Number Definition...   Code........ Heading...... .Format
Assoc..
@ID                      D       0                                        PART1
FIELD1           D       1
FIELD2           D       2
3 records listed.
>LIST DICT PART2
DICT PART2       11:08:38am 05 Dec 1994  Page 1
Type &
Field..... Field. Field........     Conversion..    Column.......   Output
Depth &
Name.....  Number Definition...         Code........ Heading....... Format
Assoc..
@ID                      D      0                                          PART2
FIELD1           D      1
FIELD2           D      2
3 records listed.
```

The following example uses LIST.INDEX to report information about the secondary indexes for FIELD1 in all the part files (PART1 and PART2 in this case) for the distributed file DIST.FILE:

```
>LIST.INDEX DIST.FILE ALL
Alternate Key Index Summary for file DIST.FILE
File........... DIST.FILE
Indices........ 2 (0 A-type, 0 C-type, 2 D-type, 0 I-type, 0 S-type)
Index Updates.. Part1: Enabled, No updates pending
                       Part2: Enabled, No updates pending
Index name    Type    Build Nulls In     DICT    S/M    Just    Unique Fld No
FIELD1  D    Not Reqd Yes No   S      L      N  1
```

DELETE.INDEX deletes the specified index of the same name in each part file where it is found, regardless of whether it is common to all part files. The next example uses DELETE.INDEX to delete the indexes for FIELD1 from all the part files for DIST.FILE:

```
>DELETE.INDEX DIST.FILE FIELD1
Removing FIELD1 from index of partfile 'PART1'!
Removing FIELD1 from index of partfile 'PART2'!
```

Use ALL to delete all the secondary index keys common to the distributed file. For example:

```
>DELETE.INDEX DIST.FILE ALL
```

Using secondary indexes from UniVerse BASIC

UniVerse BASIC provides the INDICES function and the SELECTINDEX statement for accessing secondary indexes. The SELECTINDEX statement creates a select list from secondary indexes. The INDICES function returns information about the secondary indexes in a file.

Chapter 5: UniVerse file dictionaries

This chapter explains the purpose, function, and structure of UniVerse file dictionaries. It also explains D-descriptors, A-descriptors, S-descriptors, I-descriptors, phrases, and X-descriptors.

UniVerse dictionary entries

UniVerse files usually consist of at least one data file and an associated file dictionary. A dictionary is a hashed file that defines the structure of records in a data file. Although it is possible to create a file that is not associated with a dictionary, you cannot use RetrieVe commands with such a file.

DICT.DICT file

Because UniVerse file dictionaries are themselves data files, they too are associated with a dictionary that defines the structures of all records in UniVerse dictionaries. This metadictionary is called `DICT.DICT`. It defines the fields that make up records in all UniVerse file dictionaries. The VOC file in every UniVerse account contains a pointer to the `DICT.DICT` file.

To examine `DICT.DICT` and see the field definitions for different descriptors, enter the following command:

```
>LIST DICT.DICT
```

Kinds of dictionary entry

There are five main kinds of dictionary entry. A dictionary entry can do one of the following tasks:

- Define the output format for the display of record IDs in RetrieVe queries. This required dictionary entry usually has a record ID of *@ID*, storing the record key rather than a data value.
- Define the format and location of fields for records in the associated data file. This dictionary entry is called a data descriptor.
- Define the format of a virtual field whose value is derived from fields in the associated data file or from other files.
- Define a phrase. A phrase is any part of a RetrieVe sentence that does not include a verb and a file name.
- Contain user-defined information, such as the next available record ID for the data file or a constant used in different I-descriptors. These are called X-descriptors.

UniVerse fully supports both Prime INFORMATION and Pick-style dictionary entries.

You can put dictionary entries in the VOC file rather than in the dictionary of a particular UniVerse file. This makes them available to every file in the account. For example, the standard VOC file has data descriptors that define F1 through F10, the first 10 fields of any file.

@ID record

The @ID record defines the record ID field in the data file. It is mandatory; for example, all file dictionaries must have an @ID record. @ID specifies a field location of 0 since the record ID is not really a field in the data record.

Data descriptor

A data descriptor defines the format and location of fields for records in the associated data file. D-descriptors define fields in the data files.

There are three types of data descriptor: D-descriptor, A-descriptor, and S-descriptor. They get their names from the code in field 1 that defines the type. D-descriptors usually comprise up to eight fields, plus the record ID. A-descriptors and S-descriptors comprise 10 fields, plus the record ID.

You can use D-type, A-type, and S-type formats in the same dictionary. (The A- and S-type formats can be for either D-descriptors or I-descriptors.)

I-descriptor

An I-descriptor defines a field whose value is derived from records in the associated data file or from other files. The value is the result of an expression that is evaluated at the time the query is executed. The expression is stored in the dictionary. The value itself is not stored in the data file.

There are three types of descriptor that define virtual fields:

- I-descriptor
- A-descriptor
- S-descriptor

They get their names from the code in field 1 that defines the type. I-descriptors comprise up to eight fields, the record ID, and after they are compiled, several fields containing object code.

Phrase

A phrase contains any elements of a RetrieVe sentence except the verb and the file name. Phrases specify associated fields, define output specifications, and store often-used expressions. To define a phrase in the file dictionary, specify PH in field 1 and enter the phrase in field 2.

To use a phrase in a RetrieVe sentence, include the name of the phrase in the sentence. For example, the dictionary of a `PARTS` file contains the following phrase:

```
     BOM
0001 PH
0002 BOM.PART BOM.DESC BOM.PRICE
```

You can obtain a report containing the part number, description, and price by entering the following:

```
>LIST PARTS BOM
```

Note: You can also put phrases in the VOC file. This makes the phrase available to all files in the account.

Special phrases

The following six phrases have special meaning:

Phase	Description
@ phrase	Specifies default fields for report output. You can also put default sort or selection expressions in the @ phrase. If no fields are specified in a `RetrieVe` command, or if an SQLSELECT statement uses an `*` (asterisk) to specify fields in a UniVerse file, the fields named in the @ phrase are listed by default, along with the record ID. If neither the file dictionary nor the VOC file contains an @ phrase, only record IDs are listed. Create the @ phrase as you would any other phrase.
@INSERT	Used by SQL statements. @INSERT specifies the fields into which an SQL INSERT statement inserts values if it does not explicitly name any fields. @INSERT also specifies the order in which to insert values. The order and number of values to insert must be the same as the order and number of fields in the @INSERT phrase. If @INSERT specifies fewer fields than are defined in the file dictionary, the unnamed fields remain empty.
@KEY	Used by SQL statements. @KEY specifies the column names of a table's multiple-column primary key, or the field names of a UniVerse file's multipart record ID. You can specify the character to use to separate the parts of a multipart record ID using an X-descriptor called @KEY_SEPARATOR. I-descriptors created by the user or generated by the system use the `FIELD` function to extract the columns of a multiple-column primary key or the parts of a multipart record ID. The SQL `CREATE TABLE` statement creates the @KEY phrase to let you display a multiple-column primary key or multipart record ID as separate columns or fields for meaningful output. If a UniVerse file that is not a table has multipart record IDs, you can create an @KEY phrase in the dictionary and (optionally) an @KEY_SEPARATOR X-descriptor to specify a separator other than the default (text mark). This lets users use the SQL INSERT and UPDATE statements on the multipart record IDs of a UniVerse file.
@LPTR	Specifies the default output specification for RetrieVe command output sent to the printer. Create the @LPTR phrase as you would any other phrase. If the file dictionary does not contain an @LPTR phrase, RetrieVe uses the @ phrase. If neither the file dictionary nor the VOC file contains an @ phrase, only record IDs are printed.
@REVISE	Specifies the field names that ReVise displays by default as input prompts for the data file. If you specify field names in the command that invokes ReVise, the @REVISE phrase is ignored. If the dictionary does not contain an @REVISE phrase, ReVise uses the @ phrase. If an @ phrase exists, ReVise creates an @REVISE phrase when it is first invoked on the file. This phrase contains the names of all D-type fields currently defined in the file dictionary.

Phase	Description
@SELECT	Used by SQL statements. @SELECT specifies default fields for report output from an SQL SELECT statement. If the SELECT statement uses an * (asterisk) to specify fields, the fields named in the @SELECT phrase are listed. If the dictionary does not contain an @SELECT phrase, the @ phrase specifies default output. If the dictionary contains neither an @SELECT nor an @ phrase, only record IDs are listed.

X-descriptor

An X-descriptor is a dictionary entry that stores user-defined information. An X-descriptor can store the next available record ID for the file or a constant used in a number of I-descriptors.

Define an X-descriptor in the dictionary by specifying X in field 1 and entering the information in field 2. Use the remaining fields for additional information if you need them.

If a UniVerse file has multiple-field record IDs, you can create an X-descriptor called @KEY_SEPARATOR to specify the character used to separate the values. The separator can be any character belonging to the 7-bit character set. It cannot be `ASCII NULL (CHAR(0))` or the null value. If a dictionary does not include the @KEY_SEPARATOR entry, record ID parts are separated by text marks. The @KEY_SEPARATOR record is used by SQL statements.

Structure of dictionary entries

Each field can have more than one name. This section describes the contents of these fields. The following table shows the contents of D-descriptors, which define data file fields that store data.

Field	Field name	Contents	Description
0	@ID	*record ID*	A name that identifies the field.
1	CODE	D [*description*]	Type code and an optional description of the field.
2	LOC	*field number*	The location of the field in the data file. Two unique attributes, 9998 and 9999, are available for backwards compatibility purposes with other multivalue databases. ▪ 9998 is a special attribute identifier used to produce a sequential counter for items being processed by the Retrieve sentence. ▪ 9999 is a special attribute-defining function that returns the size (in bytes) of the item being processed by Retrieve.
3	CONV	[*conversion code*]	A formula for converting data stored in internal format into external format.

Field	Field name	Contents	Description
4	NAME	[*column heading*]	A descriptive name used as a column heading. If not specified, the record ID of the D-descriptor is used.
5	FORMAT	*width* and *justification* (L, R, T, U, or C)	The width and justification of the column used when listed in a RetrieVe report.
6	SM	S M	Singlevalued field (default). Multivalued field.
7	ASSOC	[*phrase name*]	The name of a phrase (defined by another entry in the dictionary) that links multivalued fields in an association.
8	DATATYPE	[*data type*]	The SQL data type of this field, used by SQL statements.

The next table shows the contents of I-descriptors, which define virtual fields in a file. Virtual fields derive their values by processing an expression stored in field 2 of the I-descriptor.

Field	Field name	Contents	Description
0	@ID	*record ID*	A name that identifies an interpretive, or virtual, field.
1	CODE	I [*description*]	Type code and an optional description of the field.
2	EXP	*I-type expression*	The expression that, when evaluated, produces the values referenced by this field.
3	CONV	[*conversion code*]	A formula for converting data stored in internal format into external format.
4	NAME	[*column heading*]	A descriptive name used as a column heading. If not specified, the record ID of the I-descriptor is used.
5	FORMAT	*width* and *justification* (L, R, T, U, or C)	The width and justification of the column used when listed in a RetrieVe report.
6	SM	S M	Singlevalued field (default). Multivalued field.
7	ASSOC	[*phrase name*]	The name of a phrase (defined by another entry in the dictionary) that links multivalued fields in an association.
8	DATATYPE	[*data type*]	The SQL data type of this field, used by SQL statements.

The following table shows the format of A- and S-descriptor dictionary entries.

Field	Field name	Contents	Description
0	@ID	*record ID*	A name that identifies the field.

Field	Field name	Contents	Description
1	D/CODE	A [*description*] S [*description*]	Type code and an optional description of the field.
2	A/AMC	*field number*	The location of the field in the data file, or a dummy field number if field 8 contains a correlative code.
3	S/NAME	[*column heading*]	A descriptive name used as a column heading. If not specified, the record ID of the A- or S-descriptor is used.
4		C;*number*[;*number*...] D;*number*	A code that links multivalued fields in an association.
5		S M	Singlevalued field (default). Multivalued field. Used by SQL statements.
6		[*data type*]	The SQL data type of this field, used by SQL statements.
7	V/CONV	[*conversion code*]	A formula for converting data stored in internal format into external format.
8	V/CORR	[*correlative code*]	A formula that, when evaluated, produces the values referenced by this field.
9	V/TYP	L R T U	Left-justified field. Right-justified field. Wrapped text (left-justified, text breaks between words). Left-justified text (overwrites white space).
10	V/MAX	*width*	Column width used when listing the field in a RetrieVe report.

There is no difference between the way A-descriptors and S-descriptors work. You can use I-, A-, or S-descriptors as different means to achieve the same goal. For example, you can use an I-type called EXTENSION to evaluate the following expression in field 2:

```
QTY * PRICE
```

Alternatively, you can use an A-descriptor (field 1 in the dictionary contains the code A) called `EXTENSION` to evaluate the following correlative in field 8:

```
A;N(QTY) * N(PRICE)
```

However, you cannot mix A-descriptors and I-descriptors. For example, if PRICE is defined as an I-type in the dictionary, you cannot refer to PRICE within the A-type EXTENSION.

It can be helpful to use an A-descriptor as the principal field definition and to use S-descriptors as synonyms for the A-descriptor.

The following table shows the format of a dictionary entry that defines a phrase.

Field	Field name	Contents	Description
0	@ID	*record ID*	A name that identifies the phrase.
1	CODE	PH [*description*]	Type code and an optional description of the field.
2	FUNC	*phrase*	The words that make up the phrase.

The next table shows the format of X-descriptor entries, which are user-defined.

Field	Field name	Contents	Description
0	@ID	*record ID*	A name that identifies the X-descriptor.
1	CODE	X [*description*]	Type code and an optional description of the field.
2	FUNC	*user-defined information*	Can contain various information depending on the function of the X-descriptor.

Dictionary field descriptions

The following sections describe in more detail the contents of each field in a dictionary entry. The contents of the @ID field, field 1, and field 2 are the same for D-, I-, A-, and S-descriptors. They are described first, followed by:

- Descriptions of D- and I-descriptor fields 3 through 8
- Descriptions of A- and S-descriptor fields 3 through 10

@ID field

The @ID field contains the record ID of the entry in the file dictionary. The record ID can be the name of a field in the data file, whether defined by a data descriptor or an I-descriptor, or it can be the name of a phrase or X-descriptor defined in the dictionary. The @ID field is sometimes called field 0, but it is not really a field in the same sense that data fields and I-descriptor fields are, because it stores the record key rather than a data value.

Field 1

Field 1 contains the type code and an optional description of the field or phrase defined by this dictionary entry. The first one or two characters of the field must contain one of the following codes:

Code	Description
`D`	Data descriptor
`A`	Field descriptor
`S`	Field descriptor
`I`	I-descriptor
`PH`	Phrase
`X`	User-defined information

Type code entries might look like this:

```
D  last name field
I  I-descriptor used to generate total cost.
```

Field 2

The information in field 2 depends on the type of dictionary entry. You can create different definitions or views of the same data by using different data or field descriptors that specify the same location in field 2.

The following table shows the contents of field 2 for each type of entry:

If the dictionary entry is...	Field 2 contains...
Data descriptor (D-, A-, or S-descriptor)	The location, or relative position, of the field in the data file defined by this entry. The position of the first field in a data record is 1, the position of the second field is 2, and so on. Two unique attributes, 9998 and 9999, are available for backwards compatibility purposes with other multivalue databases. ▪ 9998 is a special attribute identifier used to produce a sequential counter for items being processed by the Retrieve sentence. ▪ 9999 is a special attribute-defining function that returns the size (in bytes) of the item being processed by Retrieve.
I-descriptor	An expression to be interpreted to obtain the field value. See I-descriptor, on page 64 for more information.
Correlative descriptor (A- or S-descriptor)	A dummy field number (such as 0 or 999).
Phrase or X-descriptor	The elements that make up the phrase or other user-defined information.

D- and I-descriptors

The contents of fields 1 and 2 are described in earlier sections.

Field 3 (D and I)

Field 3 is optional. It can contain a UniVerse BASIC conversion code. UniVerse processors use this code to convert input to internal format or to convert internal data to output format.

Field 4 (D and I)

Field 4 is optional. It can store a column heading. This column heading identifies the field when output is displayed or printed. If the dictionary entry does not contain a column heading in field 4, the field

name is used as the column heading. You can create multiline column headings by using a value mark (ASCII 253) to tell RetrieVe where a new line starts. Here is an example:

```
      FNAME
0001 D
0002 2
0003
0004 First^253Name
0005 10L
0006 S
0007
```

When RetrieVe displays this column, it looks like this:

```
First.....
Name......
ANTHONY
ANNE
HAROLD
.
.
.
```

Field 5 (D and I)

Field 5 specifies the output format RetrieVe is to use when displaying data in reports. The output format must specify the length of the field and the type of justification. It can optionally specify a field padding character, a formula for formatting numbers, and a mask field.

The syntax of the output format field is almost identical to the syntax of the UniVerse `BASIC FMT` function. It is also similar to the syntax of the format masks used with numeric conversion codes, but the output format specified in field 5 is not the same as the format specified by a conversion in field 3. Output format in field 5 affects only the way information is displayed or printed in RetrieVe reports; it does not perform any input conversion. When data is output, the conversion specified in field 3 is applied first, then the output format specified in field 5 is applied. Here is a summary of the UniVerse output format syntax:

```
width [ fill ] justification [ edit ] [ mask ]
```

width is the width of the column RetrieVe uses to display the data from this field. If the data is shorter than the column width, the field is padded. If the data is longer than the column width, RetrieVe displays it in a multiline format.

fill is optional. You can specify a character to be used to pad the field. If no padding is specified, the field is padded with blank spaces.

justification specifies that the data be either left-justified, right-justified, text (that is, left-justified with breaks at blanks rather than in the middle of words), or text (that is, left-justified with text that is wider than the column width being allowed to overwrite white space). There are special codes that can be used to specify justification for fields displaying exponential data.

edit is optional and can be used to format numeric data. You can specify the number of digits to be printed after the decimal point, the descaling code, and other number formatting codes for a dollar sign, commas, leading zeros suppression, and so on. The syntax of the numeric editing code is as follows:

```
[ n [ m ] ] [ , ] [ $ ] [ Z ]
```

n is the number of digits to display to the right of the decimal point.

m is the descaling factor, which moves the decimal point to the left. If *m* is not specified, a descaling factor of 4 is assumed.

The *,* (comma) specifies that commas are to be inserted every three digits to the left of the decimal point.

The *$* (dollar sign) places a floating dollar sign before the number.

Z suppresses leading zeros.

Since UniVerse uses a default descaling factor of 4, a factor of 3 moves the decimal point one place to the right, a factor of 2 moves the decimal point two places to the right, and so on. To move the decimal point to the left, you must use a descaling factor of 5 or more. For example, to specify that the decimal point be moved two places to the left, use a descaling factor of 6.

The following table shows the effect of different editing codes on two stored numbers: 233779 and 2337.79. The top line in each cell shows the effect of the editing code on 233779. The bottom line shows the effect of the editing code on 2337.79 (the decimal point is stored in the field as part of the value).

Descaling factor	**Number of digits to display after decimal**				
	empty	**0**	**1**	**2**	**4**
none	233779 2337.79	233779 2338	233779.0 2337.8	233779.00 2337.79	233779.0000 2337.7900
0	–	2337790000 23377900	2337790000.0 23377900.0	2337790000.00 23377900.00	2337790000.0000 23377900.0000
1	–	233779000 2337790	233779000.0 2337790.0	233779000.00 2337790.00	233779000.0000 2337790.0000
2	–	23377900 233779	23377900.0 233779.0	23377900.00 233779.00	23377900.0000 233779.0000
4	–	233779 2338	233779.0 2337.8	233779.00 2337.79	233779.0000 2337.7900
6	–	2338 23	2337.8 23.4	2337.79 23.38	2337.7900 23.3779
9	–	2 0	2.3 0.0	2.34 0.02	2.3378 0.0234

The following example shows valid format specification with an editing code:

```
12R26$,
```

This indicates that data should be displayed in a column 12 characters wide (12), right-justified (R), with two digits after the decimal (2), the decimal point moved two places to the left (6), a dollar sign preceding the data ($), and a comma after every third number to the left of the decimal (,). If the stored value in the field is 233779, it is displayed as $2,337.79.

mask formats the display field. Format masking allows user-specified characters to be output with numbers. The following output format includes hyphens between numbers:

R###-##-####

UniVerse displays the stored number 124121244 as 124-12-1244. The example specifies only *justification* and *mask*. You do not have to specify *width* if you specify *mask* because the length of the mask determines the width of the column. You can also enclose a mask in parentheses. If the

mask itself is to contain parentheses, the entire mask must be enclosed. For example, an American telephone number format might be specified as follows:

R((###)###-####)

Field 6 (D and I)

Field 6 specifies whether the field in the data file should be treated as containing a single value or multiple values.

For example, in a file of authors, you might have a field called NAME for each author's name. Since there is only one name per record, field 6 contains `S` for singlevalued. In the same file you might have a field called BOOKS for books written by one author. Since there can be more than one book, field 6 contains `M` for multivalued.

ReVise and RetrieVe command output for multivalued fields differs from command output for singlevalued fields. RetrieVe uses field 6 to determine whether to look for multiple values in the field. ReVise uses field 6 to prompt for multiple values when it is used to create or edit multivalued fields. Fields that are specified as singlevalued can contain multiple values, but RetrieVe treats them as a single value.

Field 7 (D and I)

Field 7 is optional. It is used only for multivalued fields. Field 7 can contain the name of a phrase, defined in the same file dictionary, which specifies an association of multivalued fields. An association is a relationship of two or more multivalued fields in a data file. There must be a one-to-one correspondence between the values (and subvalues) of the fields related by an association. RetrieVe and SQL `SELECT` display the associated values side by side in reports, even in vertical mode. ReVise processes associated fields together.

The association is defined by a phrase stored as a separate entry in the file dictionary. This association phrase specifies the names of the fields to be associated. The name of the association phrase must also be stored in the association field (field 7) of each dictionary entry defining one of the associated fields.

For example, an inventory file contains two associated fields, PRODUCT and QTY. The association phrase defined in the file dictionary looks like this:

```
      ASSOC
0001  PH
0002  PRODUCT QTY
```

The data descriptors for PRODUCT and QTY look like this:

```
       PRODUCT                    QTY
0001   D                          D
0002   6                          2
0003
0004   Item                       Quantity
0005   25T                        5R
0006   M                          M
0007   ASSOC                      ASSOC
```

Here is an example of how associated multivalued fields are displayed together on the same line:

```
>LIST ORDERS VERT
LIST ORDERS '002' '003' VERT 01:53:51pm 05 Dec 1996 PAGE 1
@ID......   002
Date..... 11/02/94
```

```
QTY TITLE.....................    LINE.AMT.
    6 WAR & PEACE                   $29.88
  4 UNIX TEXT PROCESSING           $15.92
  8 A GUIDE TO THE PICK SYSTEM     $23.84
  3 BUILDING A BETTER MOUSETRAP    $5.94
Cust.ID.. 675326196
@ID......   003
Date..... 11/03/94
QTY TITLE.....................    LINE.AMT.
    1 UNIX TEXT PROCESSING          $3.98
  2 WAR & PEACE                    $9.96
Cust.ID.. 897066635
2 records listed
```

The example shows two book orders from an ORDERS file. Each line item is made up of data from three multivalued fields: QTY, TITLE, and LINE.AMT. Because these three fields are associated, the values stored in them are displayed together on the same line, even though the report is displayed in vertical format rather than columnar.

Field 8 (D and I)

Field 8 is optional. It specifies a data type used by SQL statements.

If field 8 is empty, SQL uses the conversion in field 3 and the format in field 5 to determine the data type. In some cases, if the file is not an SQL table and you do an SQL operation against the file, you may see unexpected results. For example, if a field contains integers and the dictionary conversion code specifies `A` or `F`, UniVerse SQL treats the data as character strings. If you want to compare such data numerically or use the data in arithmetic expressions, use field 8 to specify the field's data type as INTEGER.

The following table lists and describes the values in field 8:

DATATYPE	SQL data type category
INT[EGER]	Integer
SMALLINT	Integer
DEC[IMAL] [,*n* [,*s*]]	Integer if *s* = 0, otherwise scaled number
NUMERIC [,*n* [,*s*]]	Integer if *s* = 0, otherwise scaled number
REAL	Approximate number
FLOAT [,*n*]	Approximate number
DOUBLE	Approximate number
CHAR[ACTER] [,*n*]	Character
VARCHAR [,*n*]	Character
NCHAR[ACTER] [,*n*]	National character
NVARCHAR [,*n*]	National character
BIT [,*n*]	Bit
VARBIT [,*n*]	Bit
DATE	Date
TIME	Time

A- and S-descriptors

The contents of fields 1 and 2 are described in earlier sections.

Field 3 (A and S)

Field 3 is optional. If field 3 is specified, it contains the column heading. The column heading identifies the field when output is displayed or printed. If the A- or S-descriptor does not contain a column heading in field 3, the field name is used as the column heading, as specified by the record ID of the dictionary entry. You can create multiline column headings by using a value mark (ASCII 253) to indicate where a new line starts.

The masked decimal and numeric formatting conversion codes (`MD, MR, and ML`) are used to format numeric data into currency values. For example, the code `MR2,$` formats stored numbers as dollar values. The `R` specifies right justification, the `2` specifies that two digits are to be placed at the right of the decimal point, the comma specifies that commas are to be inserted every three digits to the left of the decimal point, and the `$` places a dollar sign at the beginning of the number. If the stored value is 233779, the output value after conversion is $2,337.79.

Date conversion codes (`D`) convert dates stored in internal format to one of several output formats. For example, the code `D4/` specifies that the year be shown in four digits and that slashes be used to separate month, day, and year. If the stored date is 8485, the date when it is output is 03/25/1991.

Field 4 (A and S)

Field 4 specifies an association of multivalued fields. An association is a relationship of two or more multivalued fields in which there is a one-to-one correspondence between the values (and subvalues) of the related fields. RetrieVe commands and SQL SELECT statements display the associated values side by side in reports, even in vertical mode.

Define the association by using `C` and `D` codes in field 4 of the dictionary entries that define the fields to be associated. One of the dictionary entries must contain a `C` code and all the other associated fields must contain `D` codes. It is the `C` code that actually specifies the fields in the association. The `D` codes point back to the field whose dictionary definition contains the list of associated fields.

The syntax of the `C` code is as follows:

C;*field1 ; field2 [; fieldn] …*

field1, *field2*, and so on, are the field numbers of associated fields whose dictionary definitions contain the `D` code in field 4.

The syntax of the `D` code is as follows:

D; *field#*

field# is the number of the field whose dictionary definition contains the `C` code in field 4.

Field 5 (A and S)

Field 5 is optional. It specifies whether SQL should treat the field in the data file as containing a single value or multiple values. For example, in a file of authors, you might have a field called NAME for each author's name. Since there is only one name per record, field 5 should contain `S` for singlevalued. In the same file you might have a field called BOOKS for books written by one author. Since there can be more than one book, field 5 should contain `M` for multivalued.

This specification affects only UniVerse SQL operations. It is ignored by RetrieVe and other UniVerse processors.

UniVerse SQL treats a field defined by an A- or S-descriptor as multivalued if the descriptor does not define field 0 and either:

- Defines an association in field 4
- Contains `M` in field 5

Otherwise the field is treated as singlevalued.

Field 6 (A and S)

Field 6 is optional. It specifies a data type used by SQL statements.

If field 6 is empty, SQL uses the conversion in field 7 and the justification in field 9 to determine the data type. In some cases, if the file is not an SQL table and you do an SQL operation against the file, you may see unexpected results. For example, if a field contains integers and the dictionary conversion code specifies `A` or `F`, UniVerse SQL treats the data as character strings. If you want to compare such data numerically or use the data in arithmetic expressions, use field 6 to specify the field's data type as `INTEGER`.

The following table lists and describes the values in field 6:

SQLTYPE	SQL data type category
`INT[EGER]`	Integer
`SMALLINT`	Integer
`DEC[IMAL][,n[,s]]`	Integer if *s* = 0, otherwise scaled number
`NUMERIC[,n[,s]]`	Integer if *s* = 0, otherwise scaled number
`REAL`	Approximate number
`FLOAT[,n]`	Approximate number
`DOUBLE`	Approximate number
`CHAR[ACTER][,n]`	Character
`VARCHAR[,n]`	Character
`NCHAR[ACTER][,n]`	National character
`NVARCHAR[,n]`	National character
`BIT[,n]`	Bit
`VARBIT[,n]`	Bit
`DATE`	Date
`TIME`	Time

Field 7 (A and S)

Field 7 is optional. It can contain a BASIC conversion code. UniVerse processors use this code to convert input to internal format or to convert internal data to output format.

Field 8 (A and S)

Field 8 is optional. It can specify a correlative code that is evaluated when a RetrieVe sentence or SQL statement containing the descriptor is executed. A- and S-descriptors that contain a correlative code in field 8 are like I-descriptors in that the fields they define do not physically exist in the data file. Instead, the value of such fields is calculated from constants, data values in other fields in the data record, or data in other files.

The main difference between codes used as conversions in field 7 and codes used as correlatives in field 8 is the way in which UniVerse applies them. In general, correlatives are applied before conversions. More specifically, correlatives are applied to data immediately after it is read from a file but before it is selected, sorted, or otherwise processed. Conversions, on the other hand, are applied to constants in the command line and to data after it is processed, just before it is output.

For example, a correlative code might multiply data in the field QTY by data in the field PRICE. Then the results might be sorted. Just before it is output, a conversion might format the data by putting in the decimal point and adding a dollar sign ($).

Field 9 (A and S)

Field 9 specifies the justification of data in a report. An `R` in field 9 specifies right justification, which should be used for numeric sorting. An `L` in field 9 specifies left justification, which should be used for alphabetic sorting. A `T` in field 9 specifies text justification: text in the field is left-justified, and the lines break between words. A `U` in field 9 specifies text justification: text in the field is left-justified, and text that is wider than the column width overwrites white space.

Field 10 (A and S)

Field 10 specifies the width of the column RetrieVe uses to display the field data. If the data is shorter than the width of the column, the field is padded. If the data is longer, RetrieVe displays it in a multiline format.

Using I-descriptors

This section describes how to define I-descriptors in the file dictionary and how to compile them. I-descriptors define virtual fields. I-descriptors are evaluated when you execute a RetrieVe sentence that contains them. RetrieVe calculates the value of a virtual field from constants, values in other fields in the data record, or data in other files.

A- and S-descriptors with a correlative in field 8 work similarly to I-descriptors, but they are interpreted more slowly.

Note: The functions listed in this section are the only ones currently supported. Functions available in BASIC are independent and cannot be assumed to be available in I-descriptor expressions. Any functions not directly available in an I-descriptor expression can be accessed with an I-descriptor by calling a BASIC subroutine using the `SUBR` function.

Defining I-descriptors

To create an I-descriptor in the file dictionary, specify `I` in field 1 and define an I-type expression in field 2. The expression can be an arithmetic, logical, or string expression, a UniVerse BASIC function, an external subroutine call, or a file translation function.

When you define an I-descriptor, use fields 3 through 6 in the same way you would for data descriptors.

I-type expressions

An expression is a formula for calculating a value. I-type expressions specify how to derive information from values in fields in the current file or in another file, or from other sources such as literal data.

I-type expressions are similar to BASIC expressions. For information about UniBasic expressions, see *UniVerse UniBasic*.

You must compile an I-type expression in field 2 of an I-descriptor before it can be evaluated. During compilation the expression is checked for syntax errors. For more details, see Compiling I-descriptors, on page 91.

RetrieVe evaluates an I-type expression each time a record is accessed by the RetrieVe sentence or SQL statement containing the name of the I-descriptor.

An I-type expression can contain any of the following elements, either alone or combined by operators with other elements:

- Field names of D-descriptors and I-descriptors
- Arithmetic, relational, logical, and conditional operators
- Numeric and string constants
- *@variables*
- Substring extraction expressions
- `TRANS` function for file translation
- `TOTAL` function for accumulating totals
- `BASIC` functions and subroutines

Field names in I-type expressions

An I-descriptor can refer to another field by specifying its field name. This field name must identify a data descriptor or I-descriptor that is defined in the dictionary when the I-descriptor is compiled. When the expression is evaluated, the data in the named field of the current record is used.

If the field name refers to another I-descriptor, the I-type expression in that I-descriptor is included in the current I-type expression when the I-descriptor is compiled. An I-type expression included in this way cannot be a compound I-type expression.

For example, the TOTALSIZE I-descriptor contains the following expression:

```
RSIZE + ISIZE
```

This expression refers to two other I-descriptors: RSIZE, which contains the expression:

```
LEN(@RECORD)
```

and `ISIZE`, which contains the expression:

```
LEN(@ID)
```

TOTALSIZE works. However, the US.TOTAL.COSTS I-descriptor contains the following expression:

```
TOTAL.COSTS * TRANS(EXCHANGE.RATES, "US", 1, "X")
```

This expression refers to another I-descriptor TOTAL.COSTS, which contains the following compound expression:

```
SUM(COSTS); IF @1 > 0 THEN @1 ELSE 0
```

US.TOTAL.COSTS does not work because TOTAL.COSTS is a compound expression.

Warning: Before using one I-type expression in a second I-type expression, be sure that the second expression does not reference the first one. UniVerse cannot evaluate such an I-type expression. Nor should an I-type expression reference itself.

@variables in I-type expressions

@variables are special internal variables whose values are substituted when the expression is evaluated. I-type expressions use two kinds of @variable:

- @variables that hold intermediate values during the evaluation of compound expressions
- @variables that represent certain system information such as the system date or a field mark

The following table lists @variables that represent system information:

@variable	Value
@DATE	Internal date (for example, the date at the beginning of execution of the RetrieVe command)
@DAY	Day of the month from *@DATE* (1–31)
@FILE.NAME	Current file name (for example, the file named in the current RetrieVe command)
@FM	Field mark `(CHAR(254))`
@ID	Record ID of the current record
@IM	Item mark `(CHAR(255))`
@MONTH	Month of the year from *@DATE* (1–12)
@NB	Current BREAK level number
@NI	Current item counter
@RECORD	Current data record (for example, the record RetrieVe is currently processing)
@SM	Subvalue mark `(CHAR(252))`
@TIME	Internal time (for example, the time at the beginning of execution of the RetrieVe command)
@TM	Text mark `(CHAR(251))`
@USER.NO	User number
@VM	Value mark `(CHAR(253))`
@YEAR	Current year (2 digits, such as 97 or 98)

For a complete list of @variables, see *UniVerse BASIC*.

The following I-type expression counts the number of products supplied by a manufacturer:

```
DCOUNT(PRODUCTS,CHAR(253))
```

You can also use an @variable to supply the value mark (ASCII 253):

```
DCOUNT(PRODUCTS,@ VM)
```

@NI, the item counter

@NI returns the number of records returned by a RetrieVe command. In this example, the I-descriptor COUNTER contains the expression @NI. When listed, this returns a value indicating the order in which records are processed:

```
>LIST INVENTORY COUNTER ITEM
LIST INVENTORY COUNTER ITEM 15:05:26 06-12-98 PAGE 1
INVENTORY.                  Number      Item........
4010                                    1       PAPERCLIPS
5000                                    2       ERASER
1013                                    3       NOTEBOOK
4011                                    4       PAPERCLIPS
```

```
2004                                    5       PENCILS
2001                                    6       PENCILS
2002                                    7       PENCILS
1010                                    8       NOTEBOOK
1011                                    9       NOTEBOOK
1012                                    10  NOTEBOOK
10 records listed.
```

If the order of output is changed, the value of COUNTER for each record reflects its new position in the order:

```
>SORT INVENTORY COUNTER ITEM
SORT INVENTORY COUNTER ITEM 15:05:33 06-12-98 PAGE 1
INVENTORY.      Number      Item........
1010                    1           NOTEBOOK
1011                    2           NOTEBOOK
1012                    3           NOTEBOOK
1013                    4           NOTEBOOK
2001                    5           PENCILS
2002                    6           PENCILS
2004                    7           PENCILS
4010                    8           PAPERCLIPS
4011                    9           PAPERCLIPS
5000                    10      ERASER
10 records listed.
```

@NB, the BREAK level number

@NB has the value 0 on detail lines, the value 1 on the lowest level break lines, etc. It has the value 127 on the grand total line.

Note: Nonzero values of @NB occur only in I-descriptors for which the CALC keyword was given, since these are the only I-descriptors that are evaluated on total lines.

In this example the field TOTAL.AVG.COST displays both the total cost and the average cost in a single output column. The I-descriptor for this field looks like this:

```
         TOTAL.AVG.COST
0001 I
0002 TOTAL(COST);TOTAL(1);IF @NB = 0 THEN @1 ELSE (@1:@VM:@1/@2)
0003 MD2
0004 Total Cost}Avg Cost
0005 5R
0006 S
```

Here is a report using the TOTAL.AVG.COST field. This example also shows the total and average calculated separately.

```
LIST INVENTORY WITH ITEM = "NOTEBOOK" CALC TOTAL.AVG.COST TOTAL
COST AVG COST 15:03:25      06-12-98 PAGE 1

INVENTORY.      Avg Cost..  Cost.   Cost.
1013                 2.00           2.00    2.00
1010                 8.00           8.00    8.00
1011                10.00           10.00  10.00
1012                 7.00           7.00   7.00
                                            ====    ====
                        27.00           27.00  6.75
                        6.75
```

```
4 records listed.
```

Numeric constants and literal strings in I-type expressions

You can use numeric values in I-type expressions. You must enclose literal strings in single or double quotation marks.

Note: A string enclosed in double quotation marks must not contain double quotation marks. A string enclosed in single quotation marks must not contain a single quotation mark.

BASIC functions in I-type expressions

Many BASIC functions can be used in I-type expressions. The same syntax applies whether a BASIC function is used in an I-type expression or in a UniVerse BASIC program.

The following table lists all BASIC functions that you can use in I-type expressions.

@ function	DIVS function	INT function	OCONV function	STRS function
ABS function	DOWNCASE function	ISNULL function	OCONVS function	SUBR function
ADDS function	EBCDIC function	ISNULLS function	ORS function	SUBS function
ALPHA` function	EQS function	ITYPE function	PWR function	SUBSTRINGS function
ANDS function	EXP function	LEN function	QUOTE function	SUM function
ASCII function	EXTRACT function	LENS function	RAISE function	SUMMATION function
ATAN function	FIELD function	LES function	REPLACE function	SYSTEM function
CATS function	FIELDS function	LN function	REUSE function	TAN function
CHAR function	FIELDSTORE function	LOWER function	RND function	TIME function
CHARS function	FMT function	LTS function	SEQ function	TIMEDATE function
COL1 function	FMTS function	MATCHFIELD function	SEQS function	TRANS function
COL2 function	GES function	MOD function	SIN function	TRIM function
CONVERT function	GTS function	MODS function	SOUNDEX function	TRIMB function
COS function	ICONV function	NEGS function	SPACE function	TRIMF function
COUNT function	ICONVS function	NES function	SPACES function	TRIMS function
COUNTS function	IFS function	NOT function	SPLICE function	UPCASE function
DATE function	INDEX function	NOTS function	SQRT function	XLATE function
DCOUNT function	INDEXS function	NUM function	STATUS function	DELETE function
INSERT function	NUMS function	STR function		

For example, the INVENTORY I-descriptor, which counts the number of products supplied by a manufacturer, might look like this:

```
DCOUNT(PRODUCTS, CHAR(253))
```

PRODUCTS is a multivalued field. It is easy to count the number of products by counting the value marks, which are `CHAR 253`. You can use an @variable to indicate the value mark. For more information, see @variables in I-type expressions, on page 79.

In the same file, an I-type expression that adds the prices might look like this:

```
SUM(UNIT.PRICE)
```

The `SUM` function adds the values in the multivalued UNIT.PRICE field to produce a single value. If a multivalued field contains subvalues separated by subvalue marks `(CHAR 252)`, `SUM` adds the subvalues between each value mark `(CHAR 253)` and produces a multivalued total. The result does not contain subvalue marks.

Operators in I-type expressions

You can use operators to combine any of the expression elements described earlier to form more complex expressions.

The following table shows the operators you can use in I-type expressions. Operators are listed in order of precedence. You can use parentheses in an expression to indicated operator precedence.

Operator	Synonyms	Meaning
**	^	Exponentiation
* /		Multiplication Division
+ –		Unary plus Unary minus
+ –		Addition Subtraction
:	CAT	Concatenation
<	LT	Less than
>	GT	Greater than
=	EQ	Equal to
<>	NE > #	Not equal to
<=	LE = #>	Less than or equal to
>=	GE => #<	Greater than or equal to
MATCH	MATCHES	String matches pattern
AND	&	Performs a logical `AND` function on two expressions to produce a true (1) or false (0) result. If both expressions are true, the expression evaluates to 1 (true).

Operator	Synonyms	Meaning
OR	!	Performs a logical OR function on two expressions to produce a true (1) or false (0) result. If either expression is true, the expression evaluates to 1 (true).

The following example of an I-type expression contains arithmetic operators:

```
(SALARY+BONUS)*12
```

SALARY+BONUS is a parenthetical expression that adds two values from the named fields. The * operator multiplies the sum of the SALARY+BONUS expression by the constant 12.

The following example of an I-type expression contains a relational operator:

```
.05*PRICE*(TAXABLE EQ 'Y')
```

If the TAXABLE field contains Y, the value of the virtual field is 5% of the PRICE field; otherwise it is 0.

Conditional expressions in I-type expressions

You can use IF/THEN/ELSE expressions in I-descriptors. The syntax is as follows:

IF *test.expression* **THEN** *true.expression* **ELSE** *false.expression*

If *test.expression* is true, the value of the conditional expression is the value of *true.expression*. If *test.expression* is false, the value of the conditional expression is the value of *false.expression*. Both THEN and ELSE expressions are required by the syntax.

Here is an example of a conditional expression:

```
IF NAME LT MARKS THEN "SEC 1" ELSE "SEC 2"
```

If the value in the NAME field is before the value in MARKS alphabetically, the expression is true and SEC 1 is the value. If the value in the NAME field is equal to or after the value in MARKS, SEC 2 is the value returned by the expression.

Note: You cannot use IF with multivalued fields. Nor does IF return multivalued fields.

Compound I-type expressions

A compound expression comprises two or more I-type expressions separated by semicolons. Use compound expressions to improve execution efficiency and the readability of expressions. The value of a compound expression is the value of the last I-type expression.

Use @ (at sign) to reference an expression:

- @ references the value of the previous expression.
- @n references the *n*th expression in the compound expression. Expressions are numbered from left to right, starting with 1.

The @variable @ with no number refers to the previous simple expression, for example:

```
SUM(EXPENSES); IF @ THEN @ ELSE 'None'
```

When the value of SUM(EXPENSES) is 0, the value of the compound expression is None; otherwise, the value is the sum of EXPENSES, which is the result of the first expression.

Note: If you use @ in the first expression of a compound expression, its value is indeterminate. Similarly, the value of @n is not defined if it is used in expressions @1 through @n – 1 of a compound expression. For example, the value of @5 is indeterminate if you use it in the fourth expression.

The following example sums the values in the SALARY and BENEFITS fields, then checks the employee type. If the employee is exempt, the value of the I-descriptor is the sum of the salary and benefits; otherwise, the value is 0.

```
SUM(SALARY + BENEFITS); IF EMPL.TYPE EQ "EXEMPT" THEN @1 ELSE 0
```

Note: You cannot refer to a compound I-type expression in another I-type expression because the dictionary compiler is unable to properly evaluate it. This means that an I-type expression cannot contain the name of an I-descriptor that contains another compound expression.

Getting data from other files using the TRANS function

You can use the `TRANS` function to extract fields or records from other files for use in evaluating an I-type expression in the current file. The syntax of the `TRANS` function is as follows:

`TRANS([ DICT ] filename, record.ID, field#, control.code)`

filename is the name of the file whose data you want. The file can be remote, or it can be the current file.

record.ID is an expression that evaluates to the record IDs in the file from which to translate data. *record.ID* can be the name of a field in the current file that contains one of the following:

- Record IDs for records in the remote file or in the current file
- Expressions that evaluate to record IDs

field# can be the name or number of one field in the file from which data is to be translated, or it can be the @RECORD variable or its equivalent, –1. @RECORD returns the entire record (except the record ID) in the remote file. To check for the existence of a record, specify *field#* as 0 (the location of the record ID).

TRANS returns data only from D-type fields, not from I-type fields. If the specified field is multivalued, `TRANS` returns all values as subvalues.

If *record.ID* is multivalued, values are returned for each record (the result is multivalued). If both *record.ID* and *field#* are multivalued, subvalues are returned for each record.

control.code is a code that tells `TRANS` what to do if the translation fails. The code can be `C, V, X, or N`. It must be enclosed in quotation marks.

Code	Description
`C`	Returns the original record ID if the value in *field#* is empty or if a specified record does not exist.
`V`	Verifies successful translation. If the record does not exist or if the value in *field#* is empty, a message is printed and an empty value is returned.
`X`	Returns an empty value if a record does not exist or if the value in *field#* is empty. This is the default.
`N`	Returns the original record ID if the null value is found.

For example, the ACCOUNTS file translates the current payments from a field called AMT.PD in the PAYABLES file. The ACCOUNTS file has a field called VENDOR that contains the record IDs of records in

the PAYABLES file. The I-descriptor in the ACCOUNTS file dictionary is called PAYMENTS and looks like this:

```
     PAYMENTS
0001 I
0002 TRANS(PAYABLES,VENDOR,AMT.PD,"X")
0003 MR2,$
0004
0005 12R
0006 M
```

Use the I-descriptor in a sentence like the following:

```
>LIST ACCOUNTS PAYMENTS
```

You can also use TRANS to access information from a dictionary by specifying the DICT keyword. This can be useful for retrieving X-descriptors reserved for your use. Use this syntax:

TRANS(**DICT** *filename, record.ID, field#, control.code*)

Calling BASIC subroutines from I-type expressions

Use the SUBR function to call cataloged BASIC subroutines in an I-type expression. The syntax of the SUBR function is as follows:

SUBR(*name, [argument [,argument …]*)

name is an expression that evaluates to the name of a cataloged subroutine.

argument is an optional expression whose value is passed to the subroutine. You can pass up to 254 arguments to the subroutine.

Subroutines called by the SUBR function have a special format. The first argument to the SUBR function, which is the name of the called subroutine, does not pass a variable to the subroutine. However, the first parameter of the SUBROUTINE statement at the beginning of the subroutine source code names a variable whose value is returned to the SUBR function after the subroutine finishes execution. This variable is usually assigned in the subroutine. The second argument to the SUBR function (and all subsequent arguments) are variables that are passed to the second (and subsequent) parameters of the SUBROUTINE statement.

For example, a subroutine called by a SUBR function might look like this:

```
SUBROUTINE BENEFITS (FRINGE,X,Y)
 *
 * FRINGE IS THE ARGUMENT USED TO PASS THE RESULT BACK.
 *
 FRINGE = (X + Y) * 0.14
 RETURN
 END
```

An I-type expression that calls the BENEFITS subroutine might look like this:

```
SUBR("*BENEFITS",REGULAR,OVERTIME)
```

In this example the value of the I-type expression is the sum of the values of the REGULAR and OVERTIME fields, multiplied by 0.14. The asterisk indicates that BENEFITS is a globally cataloged subroutine.

RetrieVe may not operate correctly if a subroutine called by an I-type expression:

- Contains printer I/O statements like HEADING, FOOTING, PAGE, etc.
- Modifies @ID or @RECORD

- Relies on values saved in common from other I-type expressions

  Note: This is because the order of evaluation of I-type expressions is unpredictable.

- Updates any files RetrieVe is using for the command
- Uses `READNEXT, SELECT, or CLEARSELECT` on any select list
- Contains `STOP or ABORT`

Subroutines that handle multivalues

UniVerse BASIC has a group of special functions (called Vector Functions) and UniVerse BASIC Subroutines that handle expressions containing multivalued fields, which the standard operators and functions treat as singlevalued fields. The following table lists these functions.

The first column lists the functions for manipulating multivalued fields. The second column lists the corresponding cataloged UniVerse BASIC subroutines. The third column shows the corresponding instructions to use for singlevalued data. In this table, *m1* and *m2* represent dynamic arrays; *s1* and *s2* represent singlevalued variables; *p1*, *p2*, and so on, represent singlevalued parameters. The value of the function is the resulting dynamic array.

Vector functions	Subroutines	Single-valued fields equivalent
`ADDS` function (*m1, m2*)	–ADDS, !ADDS	*s1* + *s2*
`ANDS` function (*m1, m2*)	–ANDS, !ANDS	*s1* `AND` *s2*
`CATS` function (*m1, m2*)	–CATS, !CATS	*s1* : *s2*
`CHARS` function (*m1*)	–CHARS, !CHARS	`CHAR` (*s1*)
`COUNTS` function (*m1, p1*)	–COUNTS, !COUNTS	`COUNT` (*s1, p1*)
`DIVS` function (*m1, m2*)	–DIVS, !DIVS	*s1* / *s2*
`EQS` function (*m1*, *m2*)	–EQS, !EQS	*s1* EQ *s2*
`FIELDS` function (*m1, p1, p2, p3*)	–FIELDS, !FIELDS	`FIELD` (*s1, p1, p2, p3*)
`FMTS` function (*m1, p1*)	–FMTS, !FMTS	`FMT` (*s1, p1*)
`GES` function (*m1, m2*)	–GES, !GES	*s1* `GE` *s2*
`GTS` function (*m1, m2*)	–GTS, !GTS	*s1* `GT` *s2*
`ICONVS` function (*m1, p1*)	–ICONVS, !ICONVS	`ICONV` (*s1, p1*)
`IFS` function (*m1, m2, m3*)	–IFS, !IFS	`IF` *s1* `THEN` *s2* ELSE *s3*
`INDEXS` function (*m1, p1, p2*)	–INDEXS, !INDEXS	`INDEX` (*s1, p1, p2*)
`ISNULLS` function (*m1*)	–ISNULLS	`ISNULL` (*s1*)
`LENS` function (*m1*)	–LENS, !LENS	`LEN` (*s1*)
`LENSDP` function (*m1*)	–LENSDP, !LENSDP	`LENDP` (*s1*)
`LES` function (*m1, m2*)	–LES, !LES	*s1* LE *s2*
`LTS` function (*m1, m2*)	–LTS, !LTS	*s1* LT *s2*
`MODS` function (*m1, m2*)	–MODS, !MODS	`MOD` (*s1, s2*)
`MULS` function (*m1, m1*)	–MULS, !MULS	*s1* * *s2*
`NES` function (*m1, m2*)	–NES, !NES	*s1* \ `NE` *s2*
`NOTS` function (*m1*)	–NOTS, !NOTS	`NOT` (*s1*)
`NUMS` function *(m1*)	–NUMS, !NUMS	`NUM`(*s1*)
`OCONVS` function (*m1, p1*)	–OCONVS, !OCONVS	`OCONV` (*s1, p1*)
`ORS` function *(m1, m2*)	–ORS, !ORS	*s1* OR *s2*

Vector functions	Subroutines	Single-valued fields equivalent
SEQS function (*m1*)	–SEQS, !SEQS	SEQ (*s1*)
SPACES function (*m1*)	–SPACES, !SPACES	SPACE (*s1*)
SPLICE function (*m1, p1, m2*)	–SPLICE, !SPLICE	*s1* : *p1* : *s2*
STRS function (*m1, p1*)	–STRS, !STRS	STR (*s1, p1*)
SUBSTRINGS function (*m1, p1, p2)*	–SUBSTRINGS, !SUBSTRINGS	*s1* **[***p1, p2***]**
SUBS function (*m1, m1*)	–SUBS, !SUBS	*s1 – s2*
TRIMBS function (*m1*)	–TRIMBS	TRIMB (*s1*)
TRIMFS function (*m1*)	–TRIMFS	TRIMF (*s1*)
TRIMS function (*m1*)	–TRIMS	TRIM (*s1*)

Note: Multivalue-handling subroutines that operate on two multivalued fields at the same time result in a single multivalued field. The operation is performed on each pair of associated values: value 1 with value 1, value 2 with value 2, and so forth. If the number of values differs, the missing values are treated as empty strings, zeros, or spaces, depending on the data and the operation involved. When singlevalued fields are used in place of multivalued arguments, they are treated as multivalued fields containing only one value.

When you use a relational operator to compare a single value with several multivalues, you need the REUSE function to ensure the comparison is made for all the multivalues in the field. If you do not use REUSE, the single value is compared only to the first value of the multivalued field.

For example, the PASSES I-type expression calls the subroutine !LTS, which compares each value in the multivalued field GRADE in records in the STUDENTS file with the constant D. Any values less than D count as passes:

```
     PASSES
0001 I
0002 SUM(SUBR("!LTS", GRADE, REUSE("D")))
0003
0004 Number}Passed
0005 2R
0006 S
```

The following report includes the number of courses passed:

```
LIST STUDENTS NAME COURSE GRADE PASSES 15:50:44    06-12-98    PAGE
1
              Family                                              Number
Student ID    Name........    Courses   Grades      Passed

  123456      SMITH           PW01       A               4
                              PW02       A
                              SC01       B
                              LW01       A
  456789      JENNINGS        PW01       B               2
                              PW02       D
                              PW03       D
                              PW04       A
  345678      LAWRENCE        CS01       C               4
                              CS02       B
                              CS03       B
  234567      BROWN           ET01       D               3
                              ET02       B
                              EP01       C
```

```
                                  EP02       B

  4 records listed.
```

In the next example, an I-descriptor called RESULTS looks like this:

```
     RESULTS
0001 I
0002 SUBR("!SPLICE", COURSE, "---", GRADE)
0003
0004 Results
0005 10L
0006 S
```

Here is a report generated by the following sentence:

```
>LIST STUDENTS NAME
GIVEN RESULTS
 LIST STUDENTS NAME GIVEN RESULTS   15:50:49  06-12-98     PAGE 1
               Family         Given
 Student ID    Name........   Names.......    Results...

    123456    SMITH          JOE             PW01---A
                             ANNE            PW02---A
                                             SC01---B
                                             LW01---A
    456789    JENNINGS       GEORGE          PW01---B
                                             PW02---D
                                             PW03---D
                                             PW04---A
    345678    LAWRENCE       JEFF            CS01---C
                             SARA            CS02---B
                                             CS03---B
                                             PW01---A
    234567    BROWN          TOM             ET01---D
                             JAMES           ET02---B
                                             EP01---C
                                             EP02---B

  4 records listed.
```

Using the TOTAL function and the CALC keyword

I-descriptors use the TOTAL function to maintain running totals of other expressions. It uses these totals to evaluate the I-type expression containing the TOTAL function in order to display subtotals and totals on the final line or breakpoint lines of a report.

Note: The TOTAL function is not a UniVerse BASIC function; it can be used only in I-descriptors. Also, the TOTAL function is not the same as the RetrieVe keyword TOTAL. The TOTAL keyword accumulates and displays simple totals for numeric fields. The TOTAL function accumulates totals to be combined or manipulated within I-type expressions.

The syntax of the TOTAL function is as follows:

TOTAL(*expression*)

To use the TOTAL function:

- Create an I-descriptor that uses the TOTAL function for every field or expression for which you want to accumulate totals.
- Use the CALC keyword followed by the name of the I-type field as one of the output specifications in a RetrieVe command.

In a RetrieVe command, if you use the TOTAL function in an I-type expression and you do not use the CALC keyword, RetrieVe ignores the TOTAL function. If you use the CALC keyword with an I-descriptor that does not have a TOTAL function, RetrieVe does not generate subtotals or totals.

The values of the accumulated totals are available only on breakpoint lines or on the final line of a report. They are not available on detail lines.

When an I-descriptor is evaluated to appear on a breakpoint line, any field name of expression in the I-type expression that is not an argument of a TOTAL function contains the current value for that field or expression. This current value is the value of the field in the record and appears on the first detail line after the breakpoint. For example, the I-type expression:

```
IF VAL # 0 THEN TOTAL(COST)/TOTAL(VAL) ELSE”
```

tests only for a zero value in the current VAL field. This does not prevent the possibility that TOTAL(VAL) evaluates to 0, in which case an error is generated warning against dividing by 0. To prevent this, you can rewrite the I-type expression like this:

```
IF TOTAL(VAL) # 0 THEN TOTAL(COST)/TOTAL(VAL) ELSE ' '
```

To ensure arithmetically accurate results, remember the following equivalencies:

TOTAL(A+B)	is equivalent to	TOTAL(A) + TOTAL(B)
TOTAL(A–B)	is equivalent to	TOTAL(A) – TOTAL(B)
TOTAL(A*B)	is equivalent to	TOTAL(A) * TOTAL(B)
TOTAL(A/B)	is equivalent to	TOTAL(A) / TOTAL(B)

Note: Avoid nesting TOTAL functions, which can lead to compilation problems. Use compound expressions containing TOTAL functions instead of nesting TOTAL functions.

For example, an I-descriptor called PROFIT contains the following expression:

```
TOTAL((PRICE-COST)*QTY)
```

The CALC keyword followed by the field name PROFIT is then used as one of the output specifications in the following RetrieVe command:

```
>LIST SALES PART.NO PRICE COST CALC PROFIT
```

For each line item in the report, the PROFIT column contains values as if the PROFIT I-type expression were as follows:

```
(PRICE-COST)*QTY
```

RetrieVe keeps an accumulator for the values of (PRICE–COST)*QTY. The end of the report includes an additional report line with the accumulated total in the PROFIT column.

You can use the BREAK.ON keyword in a RetrieVe sentence like the one in the preceding example. In this case RetrieVe keeps an accumulator for both subtotals and totals for each expression in the TOTAL function. Subtotals are printed at each breakpoint, and totals are printed at the end of the report.

RetrieVe keeps as many accumulators as are necessary to produce the proper subtotals and totals for the fields that are modified with the CALC keyword. For example, an I-descriptor called MARGIN might look like this:

```
(TOTAL(PRICE-COST)/TOTAL(COST))*100
```

RetrieVe accumulates the PRICE – COST and COST values in separate accumulators and produces a subtotal or total value by dividing the PRICE – COST accumulator by the COST accumulator and then multiplying that result by 100.

Although in some cases the TOTAL function appears to work similarly to the TOTAL keyword in RetrieVe sentences, you should not confuse them. You must use the TOTAL function with the CALC keyword in cases where you want to accumulate only part of an I-type expression containing the operators *, /, **, and ^. The example of the MARGIN I-descriptor could not be replaced with the TOTAL keyword.

Generating histograms

The STR function generates a string expression that is repeated a specified number of times. This UniVerse BASIC function can be used in I-type expressions to represent values graphically as a histogram.

The syntax of the STR function is as follows:

STR(*string,repeat*)

string is the string to be generated. It can be one or more characters. *repeat* specifies the number of times to repeat the string.

In the following example, the I-descriptor GRAPH.LESSONS defines an I-type expression that uses the STR function:

```
       GRAPH.LESSONS
0001    I
0002   STR('*',LESSONS)
0003
0004
0005    50L
0006   S
```

The first expression in the function specifies an asterisk as the character to be repeated. The second expression is the name of a field which supplies the number of times the asterisk is repeated. Note that field 5 should allow for the longest string. The following command lists GRAPH.LESSONS:

```
>LIST SUN.SPORT LESSONS GRAPH.LESSONS
LIST SUN.SPORT LESSONS GRAPH.LESSONS REGISTRATION ID LESSONS
GRAPH.LESSONS.........
 85-1000                4    ****
 85-1006                4    ****
 85-1008                8    ********
 85-1016               14    **************
 85-1020                9    *********
 85-1015                3    ***
 85-1005                5    *****
 85-1007               10    **********
 85-1017                4    ****
 85-1011                4    ****
 85-1013               12    ************
11 records listed.
```

Compiling I-descriptors

Before you use an I-descriptor in a RetrieVe sentence, you should compile the expression to verify that its syntax is correct. Compilation produces object code if the syntax is correct. If the syntax is not correct, UniVerse displays an error message.

If you do not compile an I-descriptor before using it in a sentence, RetrieVe compiles it the first time it is used in a command. However, you should get in the habit of compiling I-descriptors when you create them so you can fix any syntax errors before using the I-descriptor in a command.

Note: An I-descriptor that refers to another I-descriptor is not updated when the referenced I-descriptor is changed. Therefore you should recompile all I-descriptors in a dictionary when you change any I-descriptor in that dictionary.

The compilation process is simple. Using the `COMPILE.DICT` command (or its abbreviation, CD), you can compile all the I-descriptors in a dictionary at the same time, or you can compile a specific I-descriptor. To compile all the I-descriptors in a dictionary, use the following syntax:

```
CD filename
```

To compile a specific I-descriptor, use the following syntax:

```
CD filename I.descriptor
```

When an I-descriptor is compiled, its format changes to include extra fields that contain the compiled object code and the time and date of the compilation. Depending on the type of terminal you are using, you may not be able to display compiled I-descriptors because the object code contains escape sequences that can put your terminal into an undesirable state. However, you can examine these fields using the UniVerse Editor in up-arrow mode.

Chapter 6: UniVerse file structures

This chapter explains the different kinds of files that are available in the UniVerse environment and the relationship between file type, structure, and database efficiency.

For a discussion of file design, see Creating a database, on page 46.

Static hashed files

When you create a static hashed file, separation and modulo respectively specify the group buffer size and the number of group buffers allocated for the file. You also specify a file type when you create a file. The file type determines which of several hashing algorithms should be used to distribute the records in groups. You select a file type according to the general characteristics of record IDs that will populate the file.

When you create a static hashed file, UniVerse creates a file that contains the number of groups specified by modulo. Separation is the size of group buffers in units of 512 bytes. At the same time a header block is created, the size of which is dependent on the separation. If the separation is an odd number (1, 3, 5, and so on), the header block is 1K. If the separation is an even number, the header block is the same size as the group buffer. For example, if the separation is 2, the header block is 1K. If the separation is 4, the header block is 2K, and so on.

For maximum performance, a separation higher than 16, regardless of your system's block size, is not recommended.

Dynamic files

The dynamic file (type 30) is a hashed file that resizes itself dynamically. Dynamic files automatically change modulo as the file grows or shrinks. The number of groups increases by one whenever the percentage of data in the file exceeds a specified amount (by default 80%), called the split load. The number of groups decreases by one whenever the percentage of data in the file falls below a specified amount (by default 50%), called the merge load.

You can specify additional parameters to the `CREATE.FILE` command that allow you to tune the specific characteristics of your dynamic file. The `ANALYZE.FILE` command verifies that your changes to the default parameters really increase the efficiency of the file.

To convert a static hashed file to a dynamic file, or to convert a dynamic file to a static file, use the `RESIZE` command. For some dynamic files, efficiency can be increased by changing the default parameters for the file. To change file parameters for existing dynamic files, use the `CONFIGURE.FILE` command.

Changing dynamic file parameters

The `CONFIGURE.FILE` command allows you to change dynamic file parameters for existing dynamic files. Use care when changing dynamic file parameters because you can alter the efficiency of some files.

The `DEFAULTS` keyword sets parameters to their default values. You can change individual values for MINIMUM.MODULUS, LARGE.RECORD, SPLIT.LOAD, MERGE.LOAD, and MINIMIZE.SPACE. You cannot change the values of GROUP.SIZE, RECORD.SIZE, or the hashing algorithm (`GENERAL or SEQ.NUM`).

To change GROUP.SIZE, RECORD.SIZE, or the hashing algorithm, use the `CREATE.FILE` command to create a new dynamic file with the parameters you want, then use the `COPY` command to copy the

contents of the old file to your newly created file. Use the ANALYZE.FILE command to verify the new parameters after you have changed them.

Important considerations

Dynamic files are meant to make file management easier for users. The default parameters are set so that most dynamic files work efficiently. If you decide to change the parameters of a dynamic file, keep the following considerations in mind:

- Use the SEQ.NUM hashing algorithm only when your record IDs are numeric, sequential, and consecutive. Nonconsecutive numbers should not be hashed using the SEQ.NUM hashing algorithm.
- Use a group size of 2 only if you expect the average record size to be larger than 1000 bytes. If your record size is larger than 2000 bytes, consider using a nonhashed file—type 1 or 19.
- Large record size should generally not be changed. Storing the data of a large record in the overflow buffer causes that data not to be included in the split and merge calculations. Also, the extra data length does not slow access to subsequent records. By choosing a large record size of 0%, all the records are considered large. In this case, record IDs can be accessed extremely quickly by commands such as SELECT, but access to the actual data is much less efficient.
- A small split load causes less data to be stored in each group buffer, resulting in faster access time and less overflow at the expense of requiring extra memory. A large split load causes more data to be stored in each group buffer, resulting in better use of memory at the expense of slower access time and more overflow. A split load of 100% disables splits.
- The gap between merge load and split load should be large enough so that splits and merges do not occur too frequently. The split and merge processes take a significant amount of processing time. If you make the merge load too small, memory usage can be very poor. Also, selection time is increased when record IDs are distributed in more groups than are needed. A merge load of 0% disables merges.
- Consider increasing the minimum modulo if you intend to add a lot of initial data to the file. Much data-entry time can be saved by avoiding the initial splits that can occur if you enter a lot of initial data. You may want to readjust this value after the initial data entry if you think the file may decrease in size.

Use the ANALYZE.FILE command to verify that your changes to the default parameters really increase the efficiency of the file.

Note: When a dynamic file decreases in size, the size of the file does not shrink. Instead, the unused space remains at the end of the file.

Distributed files

A distributed file is a composite hashed UniVerse file. It consists of a set of standard existing UniVerse files (including dynamic files). You use distributed files when you want to logically organize data files into a hierarchical arrangement by functional groups. Using distributed files lets you produce one summary report from individual part files. For example, you may have a part file EAST.COAST and a second part file WEST.COAST. If you want to see SALES for both files, you can define a distributed file to do this.

On UNIX systems, you can also use a distributed file when a file exceeds the size limit for UNIX files on your system, or when you want to store records in the same file but on different disk partitions.

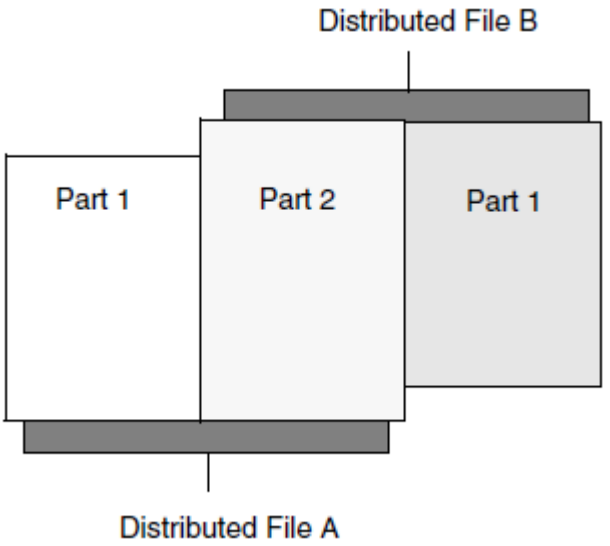
On Windows platforms, there is no file size limit.

Each UniVerse file belonging to the distributed file is a part file. Part files can reside in different accounts. A partitioning algorithm of the distributed file allocates specific record IDs to the appropriate part file.

Each part file must exist before a distributed file can be defined. A part number must be associated with each part file when a distributed file is defined. The part number is stored with each part file. The system file &PARTFILES&, found in the UV account, contains a record for each part file.

UniVerse uses a partitioning algorithm to determine the part file in which a record belongs. The partitioning algorithm must be specified when a distributed file is defined. The partitioning algorithm, or a reference to it, is stored with each part file.

A part file can belong to more than one distributed file. It must always have the same part number in each distributed file to which it belongs, and all distributed files to which a part file belongs must use the same partitioning algorithm.



The number of part files belonging to a distributed file, the partitioning algorithm, and the file parameters (such as MINIMUM.MODULUS) of each part file can be varied during the life of the file.

The partitioning algorithm

The partitioning algorithm uses part of the record ID to distribute the record to the appropriate part file. The algorithm uses the whole record ID to determine the location of the record in the part file. The partitioning algorithm must return a part number. This part number must be a nonzero integer. If a part file belongs to more than one distributed file, all distributed files to which a part file belongs must use the same algorithm.

For example, the distributed file SALES comprises two part files: EAST.COAST and WEST.COAST. The part number of EAST.COAST is 1. The part number of WEST.COAST is 2. The partitioning algorithm distributes records with IDs evaluating to 1 to part file number 1, EAST.COAST. The algorithm distributes records with IDs evaluating to 2 to part file number 2, WEST.COAST.

Creating and modifying distributed files

Use the DEFINE.DF command to create or modify a UniVerse distributed file. This command requires exclusive access to all part files of the specified distributed file. DEFINE.DF updates the &PARTFILES& file to reflect the new modifications. You must have write access to the &PARTFILES& file to create or modify a distributed file.

Creating a new distributed file

When you create a new distributed file, DEFINE.DF creates a VOC entry that defines the file. This example creates the new distributed file MY.DF:

```
>DEFINE.DF MY.DF
Creating file "D_MY.DF" as Type 30.
Added "@ID", the default record for RetrieVe, to "D_MY.DF".
Loading default SYSTEM algorithm into DICT MY.DF
Compiling "@PART.ALGORITHM".
IF INDEX ( @ID , - , 1 ) THEN FIELD ( @ID , - , 1 ) ELSE ERROR
>
```

Specifying the partitioning algorithm

You must specify the partitioning algorithm when you define a distributed file. To specify the partitioning algorithm, use one of the following:

- The SYSTEM keyword to specify a system-defined expression
- The INTERNAL keyword to specify an I-type expression
- The EXTERNAL keyword to specify an external routine

In the PIOPEN flavor you can also use the MULTIVOLUME keyword to specify multivolume files.

Using a system-defined algorithm

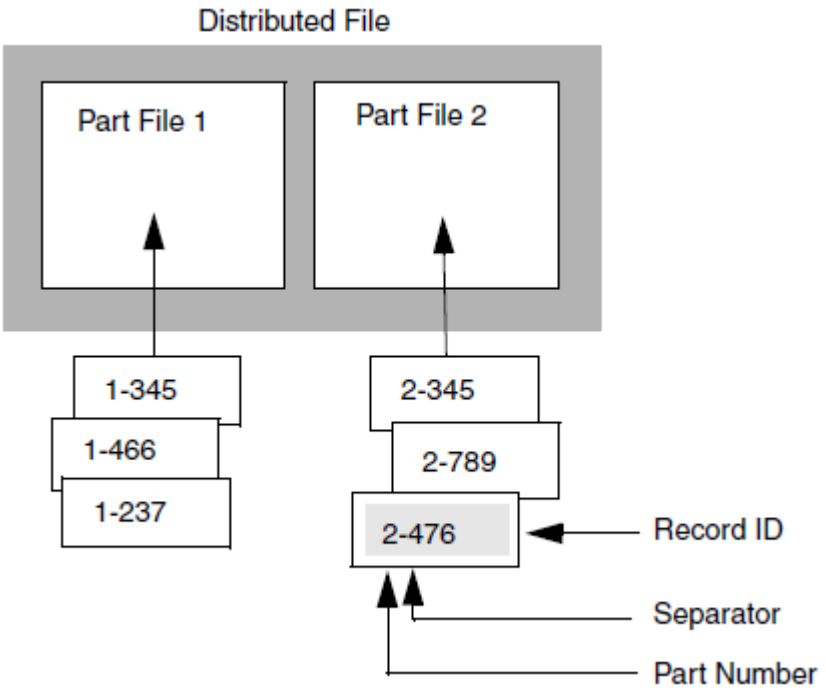
The keyword SYSTEM specifies the system-defined algorithm that UniVerse provides if you do not want to create your own. UniVerse assumes that:

- The record ID must begin with an integer.
- The record ID must contain a character separating the two parts. The separator character can be any single character except a field, value, or subvalue mark.

The following example creates the distributed file DIST.FILE, specifying that UniVerse use the default algorithm designated by the keyword SYSTEM:

```
>DEFINE.DF DIST.FILE SYSTEM
```

The system-defined algorithm uses the characters before the separator to distribute the record to the part file. The following illustration shows the record ID 2-476. The part number of the file is 2, the default separator is the hyphen, and UniVerse uses the entire value, 2-476, to locate the record in part file 2.



Consider the following example of the distributed file CUSTOMERS in the PRODUCER account. The file CUSTOMERS comprises BRITISH.MOVIES and USA.MOVIES, the two part files. The part number of BRITISH.MOVIES is 1; the part number of USA.MOVIES is 2. The partitioning algorithm (SYSTEM, in this example) distributes the records with IDs containing 1 before the separator character (for example, 1-789) to part file number 1, BRITISH.MOVIES. The algorithm distributes the records with IDs containing 2 before the separator character (for example, 2-789) to part file number 2, USA.MOVIES. The &PARTFILES& record for these two part files includes the name of the distributed file, the name of the account, the part file number, and the partitioning algorithm used with the separator character:

```
record  BRITISH.MOVIES
0001:   CUSTOMERS
0002:   PRODUCER
0003:   1
0004:   SYSTEM -

record  USA.MOVIES
0001:   CUSTOMERS
0002:   PRODUCER
0003:   2
0004:   SYSTEM -
```

Note: If each part file contains sequential record IDs, duplicate record IDs will exist. To create unique record IDs for these part files, you can edit the IDs manually or using a UniVerse BASIC program, or use `REFORMAT`.

Specifying a user-created I-descriptor algorithm

Use the INTERNAL keyword to specify a user-created I-type expression as the partitioning algorithm. The internal algorithm can be a record in the file dictionary or data file, a record in your VOC file, or an explicit part of the command when you define or modify the file. An I-type expression should use values stored in the @ID system variable to return a part number. The expression cannot access any

other data in the record. For example, the I-descriptor partitioning algorithm @ID[1,2] means that the first two characters of the record ID are to be used as the part number.

You can also use the internal algorithm with a record ID containing alphabetic characters. For example, you can create the following I-descriptor in the dictionary of the distributed file DIST.FILE:

```
IF @ID[3,4] = 'AAAA' THEN 1 ELSE IF @ID [3,4] = 'BBBB' THEN 2 ELSE 3.
```

This means that if the four characters in the record ID beginning with the third character are AAAA, the algorithm returns part number 1. If these four characters are BBBB, the algorithm returns part number 2. Any other characters return part number 3.

The following example uses `DEFINE.DF` to change a system-defined partitioning algorithm to use an internal algorithm. I.ALG is a record in the VOC file:

```
>DEFINE.DF DF DIST.FILE INTERNAL I.ALG FORCE
Warning: Partitioning algorithms for "DIST.FILE" do not match
Changing partitioning algorithm from IF INDEX( @ID, '-', 1 ) THEN
FIELD (@ID, '-', 1 ) ELSE 'ERROR' to I.ALG
Compiling "@PART.ALGORITHM".
( IF @ID [ 3 , 4 ] = AAAA THEN 1 ELSE IF @ID [ 3 , 4 ] = BBBB THEN
2
ELSE 3 )
 >
```

To specify an I-type expression as a partitioning algorithm on the command line, enclose the expression in quotation marks if it contains a space or is the same as the name of a record in the VOC file. An example is shown as follows:

```
>DEFINE.DF DF INTERNAL "IF @ID [3,4] = 'AAAA' THEN 1 ELSE IF @ID =
    'BBBB' THEN 2 ELSE 3"
Changing partitioning algorithm from to IF @ID [3,4] = 'AAAA'
THEN 1 ELSE IF @I
D = 'BBBB' THEN 2 ELSE 3 in "DF"
>
```

For more information about I-descriptors, see UniVerse file dictionaries, on page 63.

Specifying an external routine

Use the EXTERNAL keyword to specify an external partitioning algorithm. You must define parameters for the part number, record ID, and the length of the record ID. These parameters are defined in the following C programming example:

```
void part_algorithm(record_id, record_id_length, part_number)
char *record_id;
long record_id_length;
long *part_number;
{
. # External algorithm code goes here
.
.
}
```

Your external partitioning algorithm must not abort when it is passed an empty string as a record ID.

Adding a part file

`DEFINE.DF` creates a new distributed file or adds new part files to an existing distributed file. `DEFINE.DF` also creates, in the &PARTFILES& file, one record for each part file.

The following example creates a new distributed file ORDERS and its two part files, ORD.USA and ORD.CANADA. The example uses the default system partitioning algorithm.

```
>DEFINE.DF ORDERS ORD.USA 1 ORD.CANADA 2 SYSTEM
Creating file "D_ORDERS" as Type 30.
Added "@ID", the default record for RetrieVe, to "ORDERS".
Loading default SYSTEM algorithm into DICT ORDERS
Compiling "@PART.ALGORITHM".
FIELD ( @ID , - , 1 )
Part file "ORD.USA", Path "/u2/accts/ORD.USA", Part number 1 Added.
Part file "ORD.CANADA", Path "/u2/accts/ORD.CANADA", Part number 2 Added.
```

To add a third part file to this existing distributed file, you use the ADDING keyword. The new part file must be an existing UniVerse file. If the new part file is not a member of another distributed file, you must specify a unique part number. `DEFINE.DF` assigns the same partitioning algorithm used by the other part files belonging to the specified distributed file.

This example adds the third part file ORD.FRANCE to the existing distributed file, ORDERS:

```
>DEFINE.DF ORDERS ADDING ORD.FRANCE 3
Part file "ORD.FRANCE" Path "/accts/ORD.FRANCE", Part number 3 Added.
```

Adding a part file that belongs to another distributed file

If the new part file is a member of another distributed file and a part number or partitioning algorithm is specified that does not match those stored with the part file, you must use the keyword FORCE to add the part file.

UniVerse returns a message since the part number does not match the part number stored with FIRST.PART:

```
>DEFINE.DF MY.DF ADDING FIRST.PART
Part number 1 already exists in "MY.DF".
```

This next example uses FORCE to add the part file FIRST.PART that has a part number 4 to the distributed file DIST.FILE:

```
 >DEFINE.DF MY.DF ADDING FIRST.PART 4 FORCE
Warning: Part file "FIRST.PART" has a part number of 1
Compiling "@PART.ALGORITHM".
IF INDEX ( @ID , - , 1 ) THEN FIELD ( @ID , - , 1 ) ELSE ERROR
Part file "FIRST.PART", Path "/test/uv/FIRST.PART", Part number 4 Added.
```

The next example adds PART1, a part file that is a member of another distributed file, to the distributed file DIST.FILE. PART1's part number 1 and matches the part number stored in the part file.

```
>DEFINE.DF DIST.FILE ADDING PART1 FORCE
Creating file "D_DIST.FILE" as Type 30.
Added "@ID", the default record for RetrieVe, to "D_DIST.FILE".
Compiling "@PART.ALGORITHM".
IF INDEX ( @ID , - , 1 ) THEN FIELD ( @ID , - , 1 ) ELSE ERROR
Part file "PART1", Path "/test/uv/PART1", Part number 1 Added.
```

Changing the part number

The following example changes the part number for the part file ORD.MEXICO from 3 to 4:

```
>DEFINE.DF ORDERS ORD.FRANCE 4
```

```
Changing "ORD.FRANCE", Path "accts/ORD.FRANCE", from Part 3 to
Part 4.
```

Note that if the part file is a member of another distributed file and a part number is specified that does not match the part number stored with the part file, DEFINE.DF does not change the part number unless you use the FORCE keyword.

Changing the partitioning algorithm

If you want to change the partitioning algorithm of the ORDERS file to a user-defined I-type expression (stored as the record ORD.ALG in the VOCLIB file), use the INTERNAL keyword. Note that if the distributed file contains part files that belong to other distributed files, DEFINE.DF does not change the algorithm unless you add the FORCE keyword.

```
>DEFINE.DF ORDERS INTERNAL VOCLIB ORD.ALG
```

Removing part files

The following example removes the FIRST.PART part file from the distributed file MY.DF and retains the part number and algorithm in the part file. Note that UniVerse first checks &PARTFILES& to determine if the part file is a member of another distributed file. You use the RETAIN keyword to retain the part number and algorithm in the part file, for example, to restrict the file to certain keys. You cannot use the DELETE.FILE command to delete the file if you use the RETAIN keyword, because the part file still is associated with the distributed file.

```
>DEFINE.DF MY.DF REMOVING FIRST.PART RETAIN
Part file "FIRST.PART", Path "/test/FIRST.PART", Part number 4
Removed.
```

How UniVerse deletes a distributed file

You cannot delete a distributed file. Nor can you delete a file that has a part number and algorithm stored in it. If the file you want to delete is a part file, use DEFINE.DF with the REMOVING keyword to remove the part file from the distributed file. If the part file belongs to any other distributed files, you must use DEFINE.DF with the REMOVING keyword to remove the part file from those distributed files. Then you can use DELETE.FILE to delete the part file.

The following example removes the part file PART.DFA from the distributed file DFA and deletes the distributed file DFA:

```
>DEFINE.DF DFA REMOVING PART.DFA
Part file "PART.DFA", Path "/uv/PART.DFA", Part number 1 Removed.

Removing Distributed File "DFA"
DELETEd "D_DFA", Type 30.
DELETEd file definition record "DFA" in the VOC file
```

UniVerse usually deletes a distributed file automatically when there are no remaining part files. If, for any reason, this does not happen, use DELETE.FILE to delete the distributed file.

Use the CANCEL keyword to convert a part file that is no longer associated with a distributed file to a hashed file. The CANCEL keyword removes the part number and partitioning algorithm stored in the part file.

To convert the part file FIRST.PART to a hashed file, enter the following:

```
>DEFINE.DF FIRST.PART CANCEL
```

```
Removing partitioning information from file "FIRST.PART".
```

After you use CANCEL to convert the part file to a hashed file, you can use DELETE.FILE to delete the hashed file.

Listing part file information

Use the LIST.DF command to see the names and numbers of the part files that belong to a particular distributed file (without listing all the distributed files in &PARTFILES&).

Examples

Assume that you created four part files, PARTFILE1, PARTFILE2, PARTFILE3, and PARTFILE4, with part numbers 1, 2, 3, and 4, that you added to the distributed file DF1. This example shows the paths, names, and part numbers of the part files that belong to DF1:

```
>LIST.DF DF1
Part file "/rd2/john/.universe/TEST/PARTFILE2",Part number = 2.
Part file "/rd2/john/.universe/TEST/PARTFILE4",Part number = 4.
Part file "/rd2/john/.universe/TEST/PARTFILE1",Part number = 1.
Part file "/rd2/john/.universe/TEST/PARTFILE3",Part number = 3.
```

Also assume that you moved PARTFILE4 to another directory, TEST2, changed the part number of PARTFILE2 to 3, and used the CANCEL keyword on PARTFILE1 to remove the partitioning information. LIST.DF reads only the distributed file header and shows no changes:

```
>LIST.DF DF1
Part file "/rd2/john/.universe/TEST/PARTFILE2",Part number = 2.
Part file "/rd2/john/.universe/TEST/PARTFILE4",Part number = 4.
Part file "/rd2/john/.universe/TEST/PARTFILE1",Part number = 1.
Part file "/rd2/john/.universe/TEST/PARTFILE3",Part number = 3.
```

Verifying part files

Use the VERIFY.DF command to verify that the location of the part files of a particular distributed file is the same as when you created the distributed file or added the part file. VERIFY.DF also ensures that the part file numbers are nonzero integers and that the numbers are the same as when you added the part files to the distributed file. This saves you time because you need not debug your application if you have a problem. For example, if you move your part file to another partition, you may not be able to open the distributed file to which it belongs. VERIFY.DF gives you the information you need.

Example

Assume that you added PARTFILE2 with a part number of 3 to another distributed file DF2 by using the FORCE keyword. This example shows the location and the part numbers of the part files that belong to the distributed file, DF1. VERIFY.DF shows that PARTFILE4 has moved, that the part number for PARTFILE2 has changed, and that PARTFILE1 has an invalid part number.

```
>VERIFY.DF DF1
Examining "PARTFILE2".
Part number in PARTFILE2 has changed from 2 to 3.Examining "PARTFILE4".
```

```
Partfile "PARTFILE4" has been moved to/rd2/john/.universe/TEST2/PARTFILE4.
Examining "PARTFILE1".
art file "PARTFILE1" has an invalid part number of 0.
Invalid or missing Partblock for "PARTFILE1".
Examining "PARTFILE3".
```

Rebuilding distributed files

Use the REBUILD.DF command to rebuild a distributed file that has incorrect information about a path, a partitioning algorithm, or a part number that you have changed. REBUILD.DF runs in two passes. The first pass diagnoses and fixes the same problems reported by VERIFY.DF. The second pass looks for duplicate part numbers in the distributed file after rebuilding it.

REBUILD.DF removes information about a part file if an invalid part number exists, that is, a part number other than a nonzero integer. For example, if you use the keyword CANCEL with the command DEFINE.DF by mistake, you lose information about the part file. REBUILD.DF removes information about the path and the part number from the distributed file. (Use the keyword ADDING with the command DEFINE.DF to later add the part file back to the distributed file.)

Also, for example, if you move a part file to another partition, the REBUILD.DF command lets you update the information easily without using the Editor to change &PARTFILES&.

Example

This example rebuilds the information for the distributed file, DF1:

```
>REBUILD.DF DF1
SYSTEMCompiling "@PART.ALGORITHM".
IF INDEX ( @ID , - , 1 ) THEN FIELD ( @ID , - , 1 ) ELSE ERROR
Rebuilding "DF1" - First Pass.
Examining "PARTFILE2".
Part number in PARTFILE2 has changed from 2 to 3.
Fixing part number for PARTFILE2.
Examining "PARTFILE4".
Partfile "PARTFILE4" has been moved to /rd2/john/.universe/TEST2/PARTFILE4.
Fixing pathame for "PARTFILE4".
Examining "PARTFILE1".
Invalid or missing Partblock for "PARTFILE1".
Removing PARTFILE1 from DF1.
Examining "PARTFILE3".
Rebuilding "DF1" - Second Pass.
Examining "PARTFILE2".
Warning: multiple occurrences of part number 3.
Removing PARTFILE2 from DF1.
Examining "PARTFILE4".
Examining "PARTFILE3".
Warning: multiple occurrences of part number 3.
Removing PARTFILE3 from DF1.
Updating Distributed File "DF1".
Updating &PARTFILES&.
```

Type 1 and type 19 files

Type 1 and type 19 files are commonly used to store text files or UniVerse BASIC source code and listing modules. These nonhashed files are implemented as operating system directories. Each record in a type 1 or type 19 file is a separate file in that directory.

When used to store UniVerse BASIC source programs, each record contains an entire UniVerse BASIC program with newline character sequences separating each source line. Since the records in these nonhashed files are really system files, you can access them from the operating system with the standard editors or utilities. You can also access type 1 and type 19 files from the UniVerse environment using the Editor or other UniVerse processors. The UniVerse database access routines treat newline characters in type 1 and type 19 file records as field marks (ASCII 254). On input, newlines are converted to field marks, and on output, field marks are converted to newlines.

Note: The record ID for a type 1 file on Windows platforms cannot exceed 41 characters.

The following example refers to a type 1 file named BP, which is used as the default file name for the UniVerse command `RUN`. If you create BP as a type 1 file and use the Editor to create two BASIC programs called PROGRAM.1 and PROGRAM.2, UniVerse creates the following directory and files:

File or directory	Description
`BP`	A subdirectory in the current working directory.
`D_BP`	A hashed file (the file dictionary) in the current working directory.
`PROGRAM.1`	An ASCII file in the BP directory. This file contains the source lines for PROGRAM.1 separated by linefeed characters.
`PROGRAM.2`	An ASCII file in the BP directory. This file contains the source lines for PROGRAM.2 separated by linefeed characters.

When you are working in the UniVerse environment, you use the UniVerse file name and record names. For example, to compile PROGRAM.1 from UniVerse, enter the following command:

```
>BASIC BP PROGRAM.1
```

To edit PROGRAM.1 source code using the UniVerse Editor, enter the following:

```
>ED BP PROGRAM.1
```

However, to edit PROGRAM.1 source code using an operating system editor you must specify the full path. On UNIX systems, this applies to the editor vi, whether you use it from a UNIX shell environment or from UniVerse, as in the following examples:

```
$ vi BP/PROGRAM.1
>VI BP/PROGRAM.1
```

Secondary indexes

Secondary indexes are implemented as UniVerse B-tree files. You use the command `CREATE.INDEX` to create secondary indexes for a file. `CREATE.INDEX` creates a subdirectory whose name is the same as the indexed file prefixed with I_ (such as, I_ *filename*). This subdirectory, I_*filename*, contains a binary file called `INDEX.MAP` and one or more B-tree files which are the indexes themselves, one for each indexed field. The index files are named `INDEX.000`, `INDEX.001`, `INDEX.002`, and so on. `INDEX.MAP` maps the names of the files in I_*filename* to the names of the indexed fields in the UniVerse file. It also contains information used by the `LIST.INDEX` command to monitor the status of secondary indexes.

When a secondary index is created, a file definition entry is not created in the VOC file. If you want to use RetrieVe to examine the contents of a secondary index file, you must first create the following:

- A UniVerse dictionary for the index file
- A VOC entry that points to the index file

For example, if you have indexed the field CUST.ID in the ORDERS file, you would create a file dictionary for a UniVerse file called CUST.INDEX, then edit the VOC entry for CUST.INDEX to add the path of the data file (the index file):

```
>CREATE.FILE DICT CUST.INDEX
Creating file "D_CUST.INDEX" as Type 3, Modulo 1, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_CUST.INDEX".
>ED VOC
CUST.INDEX3 lines long.


----: P
0001: F
0002:
0003: D_CUST.INDEX
Bottom at line 3.
----: 2
0002:
----: R I_ORDERS/INDEX.000
0002: I_ORDERS/INDEX.000
----: FI
"CUST.INDEX" filed in file "VOC".
```

With this VOC entry in place, you can now use RetrieVe commands to gain access to the contents of the secondary index.

Multiple data files

Multiple data files can be associated with a single shared dictionary. You must use the `CREATE.FILE` command to create each data file. A UniVerse file comprising multiple data files is implemented as a directory. Each data file is a file or subdirectory in that directory, depending on whether the data file is static hashed, dynamic hashed, nonhashed, or B-tree.

In the following discussion of multiple data files, you should keep in mind that a UniVerse file is a logical entity that can comprise the following:

- A data file and its dictionary
- Multiple data files associated with a single dictionary
- A data file with no associated dictionary
- A file dictionary with no associated data files

All UniVerse commands recognize multiple data files. They are specified on the command line using the following syntax:

`filename,data.file`

filename is the name of the UniVerse file (actually a directory containing multiple data files).

data.file is the name of one of the data files. A comma separates the file name from the data file name, and no blank spaces can follow the comma.

Almost all UniVerse commands that accept a file name work with the above syntax for multiple data files.

To set up multiple data files, use the CREATE.FILE command. If the UniVerse file exists, you must specify the keyword DATA (assume that the UniVerse data file and dictionary already exist). The only change to the syntax of the CREATE.FILE command is that an optional data file name is appended to the file name. You must also specify file type (and, if needed, modulo and separation) for each data file. Here is an example setting up a new data file:

```
>CREATE.FILE   STOCK,ON.HAND
File type          = 3
Modulo             = 3
Separation         = 3
File description   = TEST
Creating a multilevel data file.
Creating file "STOCK/ON.HAND" as Type 3, Modulo 3, Separation 3.
Creating file "D_STOCK" as Type 3, Modulo 1, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_STOCK".
```

The next example shows what happens if you try to create a data file for an existing UniVerse file without using the DATA keyword:

```
>CREATE.FILE STOCK,ON.ORDER 3 3 3
"STOCK" is already in your VOC file as a file definition record.
File name =
```

Use the DATA keyword to indicate that only a data file is to be created:

```
>CREATE.FILE DATA STOCK,ON.ORDER 3 3 3
Creating file "STOCK/ON.ORDER" as Type 3, Modulo 3, Separation 3.
```

If the UniVerse file STOCK did not already exist, the following command would create a data file called ON.ORDER but would fail to create a file dictionary (D_STOCK). Using the DATA keyword assumes that the dictionary exists.

```
>CREATE.FILE DATA
STOCK,ON.ORDER 3 3 3
Creating a multilevel data file.
 Creating file "STOCK/ON.ORDER" as Type 3, Modulo 3, Separation 3.
```

If an existing UniVerse file STOCK comprises a single data file and a file dictionary when the command CREATE.FILE DATA STOCK, ON.ORDER is issued, STOCK is transformed into a file comprising two data files, STOCK,STOCK and STOCK,ON.ORDER.

A file comprising multiple data files is defined in a VOC entry. Here is a sample VOC entry for a file named STOCK comprising multiple data files, STOCK, ON.ORDER, and ON.HAND:

```
        STOCK
001:    F
002:    STOCK
003:    D_STOCK
004:    M
005:
006:
007:    ON.HAND VON.ORDERVSTOCK
008:  ON.HANDVON.ORDERVSTOCK
```

STOCK in field 2 is the relative path of the directory that contains the data files (ON.HAND, ON.ORDER, and STOCK). To create this VOC entry, each data file was created with a separate CREATE.FILE command.

Note that the record ID of this VOC file entry is also STOCK. All three data files share the same file dictionary, D_STOCK (field 3).

The file-defining entry contains an M (Multiple data files) in field 4.

Fields 7 and 8 are associated multivalued fields. Field 7 contains the UniVerse data file names in sorted order, and field 8 contains the corresponding system file names.

To access a data file using a UniVerse command, specify the file name followed by a comma and the name of the data file. For example, the following RetrieVe sentence displays a report using the data file ON.ORDER:

```
>LIST STOCK,ON.ORDER
```

Note how the following commands handle multiple data files.

Clearing multiple data files

To clear a data file, you must specify both the UniVerse file name and the data file name. The DATA keyword is unnecessary since the `CLEAR.FILE` command assumes the data file unless the DICT keyword is present. For example, because STOCK consists of multiple data files, this command clears only the data file ON.ORDER.

```
>CLEAR.FILE STOCK,ON.ORDER
```

If you do not specify the data file name `CLEAR.FILE` tries to clear a data file with the same name as the UniVerse file. The following command tries to clear a data file named STOCK,STOCK. If STOCK,STOCK does not exist, processing stops.

```
>CLEAR.FILE STOCK
```

Changing multiple data filenames

You can change the name of a data file using the TO keyword instead of a comma to delimit the old name from the new name. The following example shows the command:

```
>CNAME STOCK,ON.HAND TO STOCK,IN.STORE
```

You cannot change both the name of the UniVerse file and the name of the data file with the same `CNAME` command. You must use two separate commands:

```
>CNAME STOCK TO INVENTORY
>CNAME INVENTORY,ON.HAND TO INVENTORY,IN.STORE
```

First the name of the UniVerse file is changed from STOCK to INVENTORY, then the name of the data file ON.HAND is changed to IN.STORE.

Deleting multiple data files

If you want the operation to delete only one of the data files, you must use the DATA keyword and specify both the UniVerse file name and the data file name. If you do not use the DATA keyword, the entire UniVerse file—all data files and the file dictionary—is deleted. If you do not specify the data file name, `DELETE.FILE` attempts to delete a data file with the same name as the UniVerse file. For example, this command deletes only the data file ON.ORDER:

```
>DELETE.FILE DATA STOCK,ON.ORDER
```

In the next example, a data file name is supplied but the DATA keyword is not used:

```
>DELETE.FILE STOCK,ON.HAND
```

An error message reminds you to supply the DATA keyword.

The following command tries to delete a data file named STOCK,STOCK because the DATA keyword with no data file specified:

```
>DELETE.FILE DATA STOCK
```

If only the file name of the UniVerse file is supplied, the entire file is deleted:

```
>DELETE.FILE STOCK
Deleted file "STOCK", Type 19.
Deleted file "D_STOCK", Type 3, Modulo 1.
Deleted file definition record "STOCK" in the VOC file.
```

Long name support for multiple data files

The file creation and deletion routines allow all UniVerse files to have names up to 255 characters long. Since on certain UNIX systems the names of files and directories must be 14 characters or less, the `CREATE.FILE` command truncates the long name and adds a sequencer, using the same method used for the data or dictionary pathname. Fields 7 and 8 in the VOC entry that points to the UniVerse file map the long name used in UniVerse commands to the truncated name used for the path.

The following two commands create the paths MULTIFILE/MULTIFILE and MULTIFILE/AVERYLONG000 and a VOC entry for example:

```
>CREATE.FILE
MULTIFILE 2 1Creating file "MULTIFILE" as Type 2, Modulo 1, Separation 4.
Creating file "D_MULTIFILE" as Type 3, Modulo 1, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_MULTIFILE".

>CREATE.FILE DATA
MULTIFILE,AVERYLONGFILENAME
Creating a multilevel data file.
Creating file "MULTIFILE,AVERYLONG000" as Type 2, Modulo 1, Separation 4.
```

This is the VOC entry:

```
        MULTIFILE
 0001:  F
 0002:  MULTIFILE
 0003:  D_MULTIFILE
 0004:  M
 0005:
 0006:
 0007:  AVERYLONGFILENAMEVMULTIFILE
 0008:  AVERYLONG000VMULTIFILE
```

Pointers to files in other accounts

UniVerse has two kinds of pointers to reference files in other accounts: remote file pointers and Q-pointers. A remote file pointer uses a path to point to a file in another account. Remote file pointers are standard UniVerse VOC file entries with an F in field 1. Do not confuse them with Remote Command items, which have an R in field 1. To list the remote file pointers in your account, use the `LISTFR` command.

A Q-pointer also points to a remote file, but does so through the VOC file of another account. The Q-pointer is more portable than a remote file pointer because it points indirectly to the remote file. If the account moves in the system file structure, the Q-pointer still points to it, provided that the system administration files are correctly updated. Using Q-pointers is slightly slower than using remote file entries since the VOC file needs to be located before the files themselves are searched. UniVerse provides the SETFILE command for creating remote file pointers and the SET.FILE command for creating Q-pointers.

Creating file pointers

The SETFILE command creates a VOC file entry that points to an existing data file either in the local or in a remote account. The syntax for the SETFILE command is as follows:

```
SETFILE [ pathname ] [ filename ] [ OVERWRITING ]
```

pathname is the path of the data file for which you are creating an entry.

filename is the name used to gain access to the file from the current account. To point to a file in the same account, enter the name of the file as the path. To point to a file in a remote account, enter the relative or full pathname of the remote file.

If you use the OVERWRITING option, filename replaces an existing VOC entry with the same name.

This is the quickest way to gain access to a data file in another UniVerse account. However, if the file is moved to another directory, you can no longer reach the file because its pathname has changed.

Creating Q-pointers

Use SET.FILE to create a Q-pointer in your VOC file to a remote file. The syntax is as follows:

```
SET.FILE [ account ] [ filename ] [ pointer ]
```

account is the name of the UniVerse account containing the remote file. *account* can be specified as any of the following:

- The name of the UniVerse account as defined in the UV.ACCOUNT file. (This is the preferred way to specify the account.)
- The pathname of the directory where the remote account is located. If you use the pathname of the account, the Q-pointer loses its portability.
- The name of a login account.

filename is the name of a file in *account. pointer* is the record ID of the Q-pointer in your VOC file. If you do not specify any of the qualifiers, the SET.FILE command prompts for them.

Sizing dictionaries

Usually when you create a UniVerse file using CREATE.FILE, both the data file and its dictionary are created at the same time. UniVerse gives the dictionary the default sizing of type 3, modulo 1, separation 2 in this case.

You can also use CREATE.FILE command to specify the type, modulo, and separation for a file dictionary. This lets you determine the most efficient sizing of a dictionary. You must specify type, modulo, and separation on the command line when you invoke the CREATE.FILE command to explicitly create a dictionary. If you do not specify the size of a dictionary on the command line, the default settings are used.

The following example shows a command that creates a dictionary, specifying its size:

```
>CREATE.FILE DICT
INVENTORY 3 4 2
Creating file "D_INVENTORY" as Type 3, Modulo 4, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_INVENTORY".
```

The DICT keyword, type, modulo, and separation are specified on the command line. The next example shows the default settings that are used if no size is specified:

```
>CREATE.FILE DICT
INVENTORYCreating file "D_INVENTORY" as Type 3, Modulo 1, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_INVENTORY".
```

You can also use the `RESIZE` command to change the size of an existing dictionary.

UniVerse file protection

Since UniVerse is a multiuser system, the file handler manages several different kinds of lock which prevent problems caused by more than one user trying to gain access to the same data at the same time. There are two categories of lock: explicit locks, which are under the control of the user, and implicit locks, which are managed by the UniVerse file handler without direction from the user.

File locks and record locks

There are two kinds of explicit lock: file locks and record locks. (Record locks are also known as item locks, update record locks, and shared record locks.) Explicit locks can be set by various UniVerse BASIC statements such as `FILELOCK` statement and `READU` statement. They can also be set by various processors such as the Editor. An explicit lock prevents other users from setting the same lock or from writing to the locked file or record. Except for the exclusive use file lock used by the `RESIZE` command, an explicit lock does not prevent other users from reading the locked file or record. UniVerse BASIC statements that attempt to set a lock or to write to a locked file can either wait for a lock to be released or specify a series of statements to be executed if a lock is encountered. Subsequent attempts to set an explicit lock by the same user are successful, as are writes to a locked file or record by the same user.

File locks

A file lock prevents other users from setting a file lock on the locked file, or from setting a record lock on any record in the locked file, or from writing any record in the locked file. A file lock is set with the `FILELOCK` statement in a UniVerse BASIC program. A file lock is released by a `FILEUNLOCK` statement or by closing the file. A special exclusive-use file lock is set by the `RESIZE` command. This file lock prevents any access to the file.

Record locks

A record lock prevents other users from setting a file lock on the file containing the locked record, or from setting a record lock on the locked record, or from writing the locked record. A record lock can be set with a READL statement, READU statement, READVL statement, READVU statement, RECORDLOCK statements, MATREADL statement, or MATREADU statement. A user can write to a locked record without releasing the lock by using the WRITEU statement, WRITEVU statement, or MATWRITEU statement.

Note: UniVerse locks are case-sensitive. Because Windows does not treat file names as case-sensitive (although it is case-preserving), and because records in UniVerse type 1 and type 19 files are implemented as operating system files, it is possible for more than one user or process to access the same type 1 or type 19 file record concurrently.

Releasing locks

A record lock is released by a RELEASE statement, WRITE statements, WRITEV statement, or MATWRITE statements, or by closing the file. Record locks can also be released by the `RELEASE and UNLOCK` commands issued at the UniVerse prompt. The `RELEASE` command unlocks records locked by FILELOCK, READL, READU, READVL, READVU, MATREADL, MATREADU, and OPENSEQ statements. The `UNLOCK` command clears group, file, and record locks.

A file lock or any record locks set in a file are released when the program that set the locks terminates, provided that the file variable used to set the locks is not in a named common. If the file variable is in a named common, the locks are preserved until explicitly released. All locks are released when the user issues a `CLEARCOMMON` command or exits the UniVerse environment. For details about named common, see the BASIC COMMON statement in *UniVerse BASIC*.

Use the `LIST.READU` command to list all active group, file, and record locks.

Implicit locks

An implicit lock is set by the UniVerse file handler on a group in a file in order to preserve the integrity of the links in that group. There are three types of implicit group lock: read, write, and informational.

Read locks

A read lock is set any time any record in the group is being read. A read lock postpones any attempt to write any record in the group until all the reads have been completed. Attempts to read a record in the group are allowed.

Write locks

A write lock is set any time any record in the group is being written. A write lock prevents any other access, either read or write, to any record in the group.

Informational locks

An informational lock is set any time any record in the group has a record lock set. An informational lock is used during a write to reduce the overhead needed to check if the record being written has a record lock.

Operating system file protection

On Windows platforms, UniVerse administrators can set file permissions through the Windows User Manager and the Security menu of the File Manager.

On UNIX systems, UniVerse file permissions are set by the file creation mask umask, specified either in the user's .profile (or .login) file or in a UniVerse account's login entry.

This section provides a brief introduction to UNIX file permissions. See your UNIX documentation for complete details.

Use `umask` to specify permission bits using an octal value. This value is determined by adding any of the following values:

Octal value	Permission
400	No read by owner
200	No write by owner
100	No execute (search in directory) by owner
040	No read by group
020	No write by group
010	No execute (search in directory) by group
004	No read by others
002	No write by others
001	No execute (search in directory) by others

The first digit sets permissions for the owner, the second digit sets them for the group, and the third sets them for all other users. These values can be combined. For example, consider the following values:

Octal value	Permission
077	No read, write, execute (search) by group or other

For the first value, 040, 020, 010, 004, 002, and 001 are added to make 077. For the second value, 100, 020, 010, 002, and 001 are added to make 133.

The `umask` command specifies which bits of permission should be turned off. For example, the following command turns off bit 2, disabling write permission for users other than the owner or the owner's group for any files created after the `umask` command is executed:

```
umask 2
```

The following command makes any files created after the `umask` command is executed accessible only to the owner:

```
umask 77
```

The UniVerse `UMASK` command performs the same function as the UNIX `umask` command.

Displaying permission values

The permission values for given files can be shown using the `ls` command with the –l option. The information is shown in the following 10-position format:

```
-rwxrwxrwx
```

The first position shows the file type, as follows:

Value	Description
–	Regular file
d	Directory
c	Character special file (/*dev*)
b	Block special file (/*dev*)
l	Symbolic link

Value	Description
p	Named pipe

The remaining nine positions show read (r), write (w), and execute (x) permissions for the owner, the owner's group, and others.

The default `umask` is set to 022 (octal) so that only the owner can write to the file but all users can read it.

In addition, the owner of a file can change the permission modes for a file using the UNIX `chmod` command. Unlike `umask`, `chmod` uses the octal mask to specify which bits in the mask should be turned on. For example, the following command gives read and write permission for *file* to the owner, the group, and all other users:

```
chmod 666 file
```

`chmod` also supports symbolic arguments. The following command adds execute permission for *file* for all users:

```
chmod +x file
```

There is no UniVerse equivalent to `chmod`.

Improving file performance

This section discusses how to maintain efficient file structures. It describes file maintenance commands and tells how to repair damaged files, restore unused space, and clean up an account.

Verifying file structure

The success of hashing, and thus the speed of access to a record, depends on several related factors, including the type of records in the file; the size of the average, largest, and smallest records; and the structure of the file. By maintaining efficient file structures, you can take advantage of UniVerse's rapid data access.

UniVerse provides several commands that help you analyze file structures and fix any problems that may arise. Use these commands periodically to monitor your files. If you make changes to a file that make it significantly different from your original estimates, be sure to use these commands and make the recommended changes. The increase in performance more than makes up for any inconvenience caused by taking time to alter the file structure.

Do not forget to analyze the structure of your file dictionaries and your VOC file. Adding new I-descriptors and phrases to a dictionary affects the size of the dictionary. Adding stored sentences and paragraphs, procs, menu selectors, and so on, to the VOC file can affect performance, too, since all commands entered at the system prompt reference the VOC file. Ideally you should keep the size of VOC file entries as uniform as possible, storing longer records in files such as VOCLIB and using VOC entries as pointers to gain access to them.

Controlling hashing

There are 21 different file types. A file type is specified by one of the numbers 1 through 19, 25, or 30. Types 1 and 19 are nonhashed file types, used to contain UniVerse BASIC programs and other data organized into records that are loosely structured. Types 2 through 18 are static hashed UniVerse files. UniVerse lets you control hashing on types 2 through 18 by adjusting three factors: the file type, the modulo, and the separation. The different hashing algorithms are designed to distribute records

evenly among the groups of a file based on the characters found in the record IDs and the positions that are most significant.

To get more even record distribution and improve performance, choose a file type that is most like the keys in the file. Use the following tables as a guide.

Type	ID properties
1	A directory with records stored as files. Modulo and separation do not apply. Record names can be up to 41 characters long on UNIX and up to 255 characters on Windows platforms. (Type 1 files are most useful with files that have a few large records that are not frequently accessed. A good example of a type 1 file is a program file such as BP.)
2	Numeric and significant in the last 8 characters.
3	Mostly numeric with delimiter characters (such as * – / .) and significant in the last 8 characters.
4	Alphabetic and significant in the last 5 characters.
5	ASCII characters and significant in the last 4 characters.
6	Numeric and significant in the first 8 characters.
7	Mostly numeric with delimiter characters (such as * – / .) and significant in the first 8 characters.
8	Alphabetic and significant in the first 5 characters.
9	ASCII characters and significant in the first 4 characters.
10	Numeric and significant in the last 20 characters.
11	Mostly numeric with delimiter characters (such as * – / .) and significant in the last 20 characters.
12	Alphabetic and significant in the last 16 characters.
13	ASCII characters and significant in the last 16 characters.
14	Numeric and the entire ID should be used for hashing.
15	Mostly numeric with delimiter characters (such as * – / .) and the entire key should be used for hashing.
16	Alphabetic and the entire ID should be used for hashing.
17	ASCII characters and the entire ID should be used for hashing.
18	Arbitrary form and the entire ID should be used for hashing.
19	Type 19 files are the same as type 1 except that no subdirectories are created for long record names.
25	B-tree file. Records are stored in order by record ID.
30	Dynamic file that changes size automatically when records are added or deleted. This file type can be specified with the `DYNAMIC` keyword in place of 30.

The following table summarizes the file types according to position and contents:

Most significant	Numeric	Numeric with separators	Alphabetic	full ASCII set
Rightmost	2	3	4	5
Leftmost	6	7	8	9
Expanded	10	11	12	13

Most significant	Numeric	Numeric with separators	Alphabetic	full ASCII set
Rightmost entire key	14	15	16	17

File maintenance commands

The file maintenance commands analyze the file structure; recommend new file types, modulos, and separations; test recommended structures; and change file structures. The file maintenance commands are:

Command	Description
ACCOUNT.FILE.STATS	Gathers statistics on the state of any type of file. Use an ANALYZE.FILE command if the file is a dynamic file. If the file is hashed, use a FILE.STAT command. Use LIST.FILE.STATS to display this information.
ANALYZE.FILE	Displays the following information about a dynamic file: the hashing algorithm, the modulo, the minimum modulus, the large record size, the group size, the split load, the merge load, the current load, the number of secondary indexes, and the size of the file.
CLEAN.ACCOUNT	Performs routine maintenance on your account and corrects suspected problems with the files in your account.
CONFIGURE.FILE	Changes the parameters of an existing dynamic file.
FILE.SIZE or FILE.STAT	Displays the file type, modulo, separation and size.
FILE.USAGE	Displays a report of file use statistics.
GROUP.STAT	Displays information about the distribution of each group.
GROUP.STAT.DETAIL	Displays detailed information about the record distribution in the groups.
HASH.AID	Displays a summary of the record distribution with a new file type, modulo, and separation.
HASH.HELP	Recommends the best file type, modulo, and separation for a file.
HASH.HELP.DETAIL	Displays more detailed information than HASH.HELP and also to display the smallest, largest, and average record ID and data sizes.
HASH.TEST	Displays a summary of the record distribution with a new file type, modulo, and separation.
HASH.TEST.DETAIL	Displays a detail of the record distribution with a new file type, modulo, and separation.
LIST.INDEX	Displays information about the secondary indexes of a file.
LIST.FILE.STATS	Displays statistics about files.
RECORD	Verifies file integrity by determining the group that a record should be in.
RESIZE	Restructures a file with a new file type, modulo, and separation, or changes a file's type (such as from static hashed to dynamic hashed, or from dynamic hashed to B-tree).

Analyzing dynamic files

Use ANALYZE.FILE to display information about a dynamic file. You can specify multiple file names separated by commas on the command line. For each file specified, a report is displayed showing the following: the hashing algorithm, the modulo, the minimum modulus, the large record size, the group size, the split load, the merge load, the current load, the number of secondary indexes, and the size of the file.

The following example shows the results of using ANALYZE.FILE on the file CUSTOMERS:

```
>ANALYZE.FILE CUSTOMERS
File name                              = CUSTOMERS
 File type                                     =   2
 Number of groups in file (modulo)             =   2
 Separation                                    =   2
 Number of records                             =  10
 Number of physical bytes                      =  3072
 Number of data bytes                          =  972
 Average number of records per group           =  5.0000
 Average number of bytes per group             =  486.0000
 Minimum number of records in a group          =  4
 Maximum number of records in a group          =  6
 Average number of bytes per record            =  97.2000
 Minimum number of bytes in a record           =  76
 Maximum number of bytes in a record           =  120
 Average number of fields per record           =  3.0000
 Minimum number of fields per record           =  3
 Maximum number of fields per record           =  3
 Groups    25%    50%    75%    100%    125%    150%    175%    200% full
             0      1      1       0       0       0       0        0
```

The STATISTICS keyword causes additional information to be displayed in the report for each file. The data displayed with the STATISTICS keyword takes additional time to compile. While the data is compiled, a progress report appears. After the data compiles, you are asked to press Return to continue. The second page of the report contains the information from the standard report and also displays the following totals: number of records, large records, and deleted records; total size of record data and of record IDs; and the total amount of used and unused space.

The third page of the report gives information about records and groups. It displays the average, minimum, maximum, and standard deviation per group for the following categories: group buffers, records, large records, deleted records, data bytes, record ID bytes, unused bytes, and total bytes. This page also displays the average, minimum, maximum, and standard deviation per record of the number of data bytes, record ID bytes, and total bytes.

If you specify the NO.PAGE keyword, the report will not stop for page-break messages. Print the report using the LPTR keyword.

Listing statistics for static hashed files

Use FILE.STAT to find the current file type, modulo, separation, number of bytes, number of records, and number of groups for a static hashed file. FILE.STAT also displays the averages, the minimum and maximum number of bytes, and the minimum and maximum number of fields in a record.

The following example shows the results of using FILE.STAT on the file ORDERS:

```
>FILE.STAT ORDERS
```

```
File name                                = ORDERS
 File type                                     = 2
 Number of groups in file (modulo)             = 2
 Separation                                    = 2
 Number of records                             = 21
 Number of physical bytes                      = 3072
 Number of data bytes                          = 1120


Average number of records per group            = 10.5000
 Average number of bytes per group             = 560.0000
 Minimum number of records in a group          = 10
 Maximum number of records in a group          = 11


Average number of bytes per record             = 53.3333
 Minimum number of bytes in a record           = 48
 Maximum number of bytes in a record           = 60


Average number of fields per record            = 5.0000
 Minimum number of fields per record           = 5
 Maximum number of fields per record           = 5


Groups  25%    50%    75%    100%    125%    150%    175%    200%    full
                 0       0       2       0       0       0       0       0
```

Look first at the distribution of records into groups shown on the bottom two lines of the display. These lines show the number of groups that are a given percentage full. This file has two groups at 75% full. Percentages greater than 100 indicate that the specified number of groups have overflow buffers allocated to them. Try to avoid this condition. If any groups are over 100% full, you should change the file structure.

In general, if you expect a file to grow, groups should be clustered in the 50% full range. If you expect the file to remain about the same size, the groups should be clustered in the 75% range.

Use the rest of the report to help determine a new modulo and separation.

Analyzing any file type

`ACCOUNT.FILE.STATS` and `LIST.FILE.STATS` let you gather and display statistics on the current state of any selected file. Thus, you can check the efficiency of all the files in a UniVerse account at one time instead of issuing separate commands on each file.

`ACCOUNT.FILE.STATS` with the LOCAL option stores the statistics in the local `STAT.FILE` file. If a selected file is nonhashed, all records are selected, read, and statistics are written to `STAT.FILE`. If a selected file is dynamic, an `ANALYZE.FILE` command is executed, and the statistics are written to `STAT.FILE`. If a file is hashed, a `FILE.STAT` command is issued, and the relevant statistics are written to `STAT.FILE`.

After issuing the `ACCOUNT.FILE.STATS` command, `LIST.FILE.STATS` lets you display the statistics.

```
            * * *  F I L E     S T A T S     R E C O R D     Page 1
 /u1/uv gathered on 05 Dec 1994 at 14:09:21.


                            Record. File......
File Name.... Type Mod. Sep Count.. Size (ext)  25%  50%  75%
```

```
>100%

D_&COMO&        18    1    1    1     1536       1     0     0
&COMO&          1               17       34330
D_&DEVICE&      3     1    2    20    3072       0     0     0
1
&DEVICE&        2     3    4    15    8192       2     0     1
D_&ED&          3     1    2    1     2048       1     0     0
&ED&            1               9     757
D_&HOLD&        3     1    2    1     2048       1     0     0
&HOLD&          1               1     286
D_&PH&          3     1    2    1     2048       1     0     0
&PH&            1          2          61080
D_&SAVEDLISTS&  3     1    2    1     2048       1     0     0
&SAVEDLISTS&    1               33    123803
D_&TEMP&        3     1    2    1     2048      1     0     0
&TEMP&          1               8     199
Press any key to continue....
```

File usage statistics

You can collect file usage statistics on any hashed file as long as you have write permission. To start gathering usage statistics for a file, you must first issue the `FILE.USAGE.CLEAR` command.

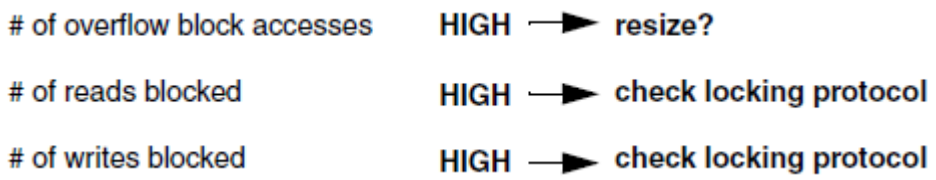
Then issue `FILE.USAGE` to show how the files in your application are being accessed and to indicate any decreased performance due to frequent access of records in an overflow block. Here is sample output from the `FILE.USAGE` command:

```
>FILE.USAGE VOC
Usage statistics for file VOC
        Reads for update =    9369      77.56%
        Blocked read           =     0      0.00%
        Read sharelocks      =     0      0.00%
        Blocked sharelocks   =     0      0.00%
        Oversized reads      =     0      0.00%
        Total reads           12080
        Writes for update    =     0      0.00%
        Update write         =   932      99.97%
        Blocked write        =     0      0.00%
        Oversized writes        =     0      0.00%
        Total writes            =  9325
        Total selects           =   218
        Total deletes           =     2
        Total clears            =     0
        Total opens             =     0
        Overflow buffer scans   =    66
        Buffer compactions      =     0
```

In this example, the report shows that this file is not experiencing any lock bottlenecks, since the blocked reads and writes are zero.

The `FILE.USAGE` command lets you see whether the system is experiencing performance degradation due to excessive overflow block accesses. The standard approach to file sizing is to have no overflow blocks and to have a high ratio of data bytes to physical bytes. Some files cannot be sized in this ideal way because of a wide range of record sizes. The `FILE.USAGE` command helps determine whether a file sizing that is efficient in terms of disk space is acceptable in terms of performance. If the number of oversized block accesses (for read or write) is low, the performance impact of the sizing is negligible.

If a significant percentage of reads or writes are blocked, you may want to examine the locking protocol in your application. Figure shows the steps you can take to improve the performance degradation due to blocked accesses. Perhaps you can reduce the number of blocked access attempts by creating additional lock records that separate concurrency control where possible. Or you may be able to reduce the amount of time any one user can lock a record by releasing it sooner in the application code.



The statistics for a file are reset by the `FILE.USAGE.CLEAR` command, or by a `RESIZE` operation. Once you have gathered use statistics on a file, you can improve performance by using the `FILE.USAGE.OFF` command to turn off statistics gathering.

Showing how records are distributed in groups

The `GROUP.STAT` command gives another view of the distribution of records into groups. It displays the total number of bytes in each group, the number of records in each group, and a histogram of the total number of records in each group. The histogram makes it very easy to spot uneven distribution of records among groups.

Analyzing the ORDERS file with `GROUP.STAT` produces the following report:

```
>GROUP.STAT ORDERS
Type description= Hashed, keys end in numbers.
Bytes Records   File= ORDERS Modulo= 2  Sep= 2  Type= 2
  520     10 >>>>>>>>>>
  600     11 >>>>>>>>>>>
======   ====
 1120     21   Totals
  560     10   Averages per group
   40      0   Standard deviation from average
  7.1    0.0   Percent std dev from average
```

The histogram shows that the distribution of records into groups is fairly even. If the histogram for a file shows an uneven record distribution, you should consider restructuring the file.

`GROUP.STAT.DETAIL` gives a detailed breakdown of the allocation of records into groups. The report lists each record by size, record ID, and group:

```
>GROUP.STAT.DETAIL ORDERS
Type description= Hashed, keys end in numbers.
Bytes  Record.id  File= ORDERS Modulo= 2 Sep= 2 Type= 2
   48  102
   48  204
   60  210
   56  318
   48  504
   52  112
   48  202
   56  418
   48  502
```

```
   56  704
------
  520 Bytes     10 Records in Group 1


   52  111
   48  501
   52  103
   56  301
   56  319
   52  101
   56  113
   52  203
   60  401
   60  419
   56  605
------
  600 Bytes     11 Records in Group 2


Bytes   Records
 1120      21   Totals
  560      10   Averages per group
   40       0   Standard deviation from average
  7.1     0.0   Percent std dev from average
```

This report lets you see if a single large record is causing a group to overflow or if there are too many records in a group. A single large record can cause a group to overflow, even if the file is efficiently hashed.

Determining the best file structure for hashed files

Once you have determined that a file needs to be restructured, use `HASH.HELP` and `HASH.HELP.DETAIL` to have the system recommend hashing for the file. `HASH.HELP` examines every record ID and every record in an existing file and recommends a file type, modulo, and separation.

Here is an example of the display produced by `HASH.HELP`:

```
>HASH.HELP ORDERS
File ORDERS Type= 2  Modulo= 2 Sep= 2 12:37:43pm 05 Dec 1994
PAGE  1
Of the 21 total keys in this file:

    21  keys were wholly numeric (digits 0 thru 9)
        (Use File Type 2, 6, 10 or 14 for wholly numeric keys)

    0   keys were numeric with separators (as reproduced below)
        0123456789#$%&*+-./:;_
        (Use File Type 3, 7, 11 or 15 for numeric keys with
separators)

    0   keys were from the 64-character ASCII set reproduced
below
        !"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_'
       (Use File Type 4, 8, 12 or 16 for 64-character ASCII
keys)

    0   keys were from the 256-character ASCII set
```

```
            (Use File Type 5, 9, 13 or 17 for 256-character ASCII
keys)

The keys in this file are more unique in their right-most eight bytes.
The smallest modulo you should consider for this file is 3.
The smallest separation you should consider for this file is 1.
The best type to choose for this file is probably type 2.
```

HASH.HELP.DETAIL produces the same report as HASH.HELP with added information on the total number and length of keys in the file, the lengths of the smallest and largest keys, the length of the average key, the total number of records and bytes in the file, the lengths of the smallest and largest records, and the average number of bytes in a record.

The recommendations produced by HASH.HELP and HASH.HELP.DETAIL are just that—recommendations. They are a good starting point for testing possible file structures. Since the recommendation is based on an average of the existing records, the recommended modulo is the smallest modulo you should consider and may be lower than optimal.

Testing a new file structure

If you decide that the file structure needs to be altered, you can test the effect of different values for file type, modulo, and separation without actually altering the file. The HASH.AID, HASH.TEST, and HASH.TEST.DETAIL commands let you experiment with various values for the three parameters. Each command produces a different report.

HASH.AID produces the same report as FILE.STAT with statistics for a hypothetical file type, modulo, and separation. HASH.AID also saves some summary data in HASH.AID.FILE, which you can use to quickly compare possible file sizes. HASH.TEST and HASH.TEST.DETAIL produce the same kind of report as GROUP.STAT and GROUP.STAT.DETAIL.

When you test values for file type, modulo, and separation, start with the values recommended by the HASH.HELP and HASH.HELP.DETAIL commands. If the results leave the record distribution unbalanced or the groups too full, test other values. Remember that the modulo should be a prime number. Use the PRIME command to determine the prime number closest to the estimated number of groups you need for your file.

Resizing files

Once you have determined the best file type, modulo, and separation for a file, you can restructure the file by using the RESIZE command. If you want to keep an existing file type, modulo, or separation, enter an asterisk (*) as its value. For example, to change the VOC file's modulo to 2 but leave the file type and separation the same, enter the following:

```
>RESIZE SUN.MEMBER * 2 *
```

RESIZE changes the physical structure of the file on the disk. RESIZE creates a file of the specified size and copies records from the original file to the new file. The file is locked to prevent other users from accessing it while it is being resized.

To give others users access to a file while it is being resized, specify the CONCURRENT option.

If you do not have enough disk space to create the new file, the file can be resized in place using the INPLACE option. When you use the INPLACE option, the file is resized in the same physical location on the disk by removing all oversized records from the file.

Use the USING keyword to specify the path of a directory to be used as workspace for the temporary files that are created while the file is being resized.

After the file has been resized, the temporary files are copied into the newly resized file, then deleted.

Note: When you decrease the size of a UniVerse file using RESIZE with the INPLACE option, the size of the file does not shrink. Instead, the unused space is moved to the end of the file. In some cases the size of the file may actually increase because of workspace allocated by RESIZE.

Changing file type

Use the RESIZE command to change a file's type. You can change a static hashed file to a dynamic hashed file, or either of these to a B-tree or other nonhashed file (type 1 or type 19). Change the file's type to a dynamic file for easier file management for files that change size frequently. Change the file's type to a B-tree file to improve access to stored data. Change the file's type to a type 1 or type 19 file to store text, program source code, or other data that does not have much structure to it. RESIZE uses the same syntax as it does when converting one type of hashed file to another. RESIZE takes the name of the file from the command line or from an active select list and changes the file type to the one you specify.

When using RESIZE to change a static hashed file, B-tree file, or type 1 or type 19 nonhashed file to a dynamic hashed file, you can specify dynamic file parameters that are different from the defaults. You can set the following parameters:

- SEQ.NUM
- GENERAL
- GROUP.SIZE
- MINIMUM.MODULUS
- SPLIT.LOAD
- MERGE.LOAD
- LARGE.RECORD
- RECORD.SIZE
- MINIMIZE.SPACE

RESIZE cannot change the parameters on an existing dynamic file. Use the CONFIGURE.FILE command to change an existing dynamic file's parameters. For more information, see Changing dynamic file parameters, on page 92.

Monitoring the VOC file

Since the VOC file is the central file in your account, its size probably changes frequently. You should regularly monitor it using FILE.STAT, GROUP.STAT, and the other commands discussed in this chapter.

Repairing damaged files

Occasionally a system crash or other unexpected event damages a file. When this happens, you may not be able to gain access to the file. Sometimes it is possible to repair the damage using the RESIZE command.

To try to repair the file, use the following syntax (the asterisks keep the file type, modulo, and separation the same):

```
RESIZE filename * * *
```

Cleaning up an account

It is important to monitor the file structure to ensure that each file has the most efficient structure. Efficient file structures ensure fast access and make UniVerse work at peak performance levels. It is also important to monitor your account structure for file structure problems and old temporary files. UniVerse provides a CLEAN.ACCOUNT command for this purpose.

CLEAN.ACCOUNT performs routine maintenance activities on your account and can correct suspected problems with files. If you do suspect problems, notify your system administrator and use this command with care.

CLEAN.ACCOUNT searches for special temporary files such as &TEMP& and &PH&. It deletes the &TEMP& file and clears the &PH& file if you give it permission. It also searches the &SAVEDLISTS& file for temporary records and deletes them.

After examining the special files, CLEAN.ACCOUNT examines all the files defined in the VOC file. If it discovers any problems, it reports them on your terminal.

```
>CLEAN.ACCOUNT
Do you wish to delete the file "&TEMP&" (Y/N) ? Y
Now deleting file "&TEMP&".

DELETEd "&TEMP&", Type 19.
DELETEd file "D_&TEMP&", Type 3, Modulo 1.
DELETEd file definition record "&TEMP&" in the VOC file.

Now selecting all the File definition records in your VOC file.

Now selecting all the UNIX files in your directory.

Now processing the VOC file definition records selected.

Processing file definition record "GLOBAL.CATDIR"
Processing file definition record "NEWACC"
Processing file definition record "UNIVERSE.STAT.FILE"
Processing file definition record "UV_SCHEMA"
Processing file definition record "APP.PROGS"
Processing file definition record "APP.PROGS.O"

File "APP.PROGS.O" points to a remote file that I cannot open
     DATA file pathname = "/doc/uv/APP.PROGS.O".*
Do you wish to delete the pathname from the record "APP.PROGS.O" (Y/N)? Y
```

Chapter 7: File triggers

This chapter describes triggers, which in UniVerse BASIC are programs associated with files. UniVerse executes (“fires”) triggers when some action changes the file’s data.

Note: UniVerse SQL also supports file triggers. For more information, see *UniVerse SQL Reference*.

Applying business rules

Developers use triggers to enforce certain business rules when users or programs make changes to a database. Traditional database systems require that applications enforce their own business rules. When many applications reference the same files, they need duplicate code to enforce business rules for those files, making it difficult to maintain consistency. Databases that use triggers, on the other hand, can enforce business rules directly. When the database enforces business rules, it enforces them consistently, and you need only maintain one single code source.

Some business rules that application programs enforce are:

- A customer number on an order must correspond to an existing customer.
- The program cannot delete customers if they have any orders.
- When a customer’s number changes, the program updates all of the customer’s orders.

Triggers can enforce more precise rules, such as the following:

- A customer number on an order must correspond to an active customer.
- The program cannot delete customers if they have any open orders.
- Certain changes are not allowed, depending on date, time, user, or specific data.
- The program inserts date, time, and user stamps.
- The program maintains audit copies of changes to another table.

Using triggers

UniVerse SQL statements, UniVerse BASIC programs, ProVerb procs, and UniVerse paragraphs can all change a file’s data, and thus fire a trigger program. Events that change the database include:

- `INSERT` statement
- `UPDATE` statement
- `DELETE` statement
- `BASIC WRITE` statements
- `BASIC DELETE` statements
- ProVerb `F-WRITE` command

Making changes to the database using the UniVerse Editor and ReVise also fire triggers. Other UniVerse and operating system commands and operations, such as `CLEAR.FILE`, *rm*, or a roll-forward, do not.

When does a trigger fire?

A trigger can fire either before or after a change is made to the database. When you create a trigger, you specify whether it should fire before the event that changes the database or after it. A trigger program fires (executes) for each record that UniVerse inserts, updates, or deletes.

Note: UniVerse supports only triggers that fire once for each record. It does not support triggers that fire one for each statement.

What events fire a trigger?

UniVerse treats all events that change the database as one of the following:

- `INSERT` statement
- `UPDATE` statement
- `DELETE` statement
- UniVerse BASIC `WRITE` statement
- UniVerse BASIC `DELETE` statement

UniVerse treats a UniVerse BASIC WRITE statement as an INSERT statement if the record ID does not exist. It treats it as an UPDATE statement if the record ID already exists. To change a primary key using UniVerse BASIC, you must first `DELETE` the old record (firing any `DELETE` triggers), then WRITE a new record (firing any `INSERT` triggers).

If you use UniVerse SQL to change a primary key, any `BEFORE UPDATE` triggers fire during the automatic `DELETE`, and any `AFTER UPDATE` triggers fire during the automatic WRITE.

Creating triggers

You create triggers for a file using the CREATE TRIGGER statement. You delete them using the DROP TRIGGER statement. By default, UniVerse enables triggers when you first create them.

You must have write permissions on a file to create or drop a trigger.

The syntax options for creating triggers are described in the following sections.

UniVerse BASIC WRITE on existing record

To create a trigger for an existing record using a UniVerse BASIC WRITE statement, enter the following command:

`CREATE TRIGGER` *`trigger_name`* `[BEFORE | AFTER]` **`UPDATE`**

UniVerse BASIC WRITE of a new record

To create a trigger for a new record using a UniVerse BASIC WRITE statement, enter the following command:

```
CREATE TRIGGER trigger_name [BEFORE | AFTER] INSERT
```

UniVerse BASIC DELETE of existing record

To create a trigger when deleting an existing record using the UniVerse BASIC DELETE statement, enter the following command:

```
CREATE TRIGGER trigger_name [BEFORE | AFTER] DELETE
```

Modifying triggers

If you want to modify a trigger program, do the following:

- Use the DROP TRIGGER statement to drop all triggers that use the program.
- Change, compile, and catalog the trigger program.
- Use the CREATE TRIGGER statement to re-create the trigger.

Listing information about triggers

Use the `LIST.SICA` command to list information about a file's triggers.

Note: For information about `LIST.SICA` against a UniVerse table, see *The UniVerse SQL Reference*.

The `LIST.SICA` command returns the following information:

- Names of triggers
- Names of the UniVerse BASIC trigger programs they invoke
- Whether the triggers are enabled or disabled

The following example assumes the trigger programs FILETRIG has been globally cataloged before executing the CREATE TRIGGER statements.

```
SUBROUTINE
SQLTRIG(SQLTRIG,SCHEMA,TABLE,EVENT,TIME,NEWID,NEWREC,OLDID,OLDREC,ASSOC,ASS
OC.EVENT,COUNT,CHAIN.CASCADE,CASCADE)
COMMON/SQLTRIG/ AUDITFV
FILEOPENED = FILEINFO(AUDITFV,0)
IF FILEOPENED NE 1 THEN
    OPEN '','TRIG_AUDIT' TO AUDITFV ELSE CRT 'COULD NOT OPEN TRIG_AUDIT' ;
RETURN
END
ITEM = 'UPDATED ':OLDID:' IN ':TABLE
ID = TIMEDATE(): '*' :OLDID
LOOP
    RECORDLOCKU AUDITFV,ID
        ON ERROR RETURN
    LOCKED
        NAP 250
        CONTINUE
    END
    EXIT
REPEAT
WRITE ITEM ON AUDITFV,ID
```

```
RETURN
```

Trigger programs

Trigger programs are compiled and cataloged UniVerse BASIC subroutines. You must catalog such programs either normally or globally. For information about cataloging UniVerse BASIC programs, see *UniVerse BASIC*.

Each UniVerse BASIC subroutine must define 14 arguments, in the following order:

Argument	Contains
trigger.name	Name of the trigger.
schema	Name of the schema containing the trigger's file.
file	Name of the trigger's file.
event	`INSERT`, `UPDATE`, or `DELETE`.
time	`BEFORE` or `AFTER`.
new.recordID	If *event* is `INSERT` or `UPDATE`, new record ID, otherwise empty.
new.record	If *event* is `INSERT` or `UPDATE`, new record, otherwise empty.
old.recordID	If *event* is `UPDATE` or `DELETE`, old record ID, otherwise empty.
old.record	If *event* is `UPDATE` or `DELETE`, old record, otherwise empty.
association	Name of a dynamically normalized association.
association.event	`INSERT`, `UPDATE`, or `DELETE` on a dynamically normalized association.
count	Number of triggers that are current firing.
chain.cascade	Number of active triggers since the last cascade operation.
cascade	`UPDATE` or `DELETE` if a cascaded update or delete fires the trigger, otherwise empty.

The position of the argument determines what information it contains., For the syntax of the `SUBROUTINE` statement, see *UniVerse BASIC*.

UniVerse BASIC programs can use the *newrecordID*, *new.record*, *old.recordID*, and *old.record* variables to access the current record. For `INSERT` and `UPDATE` events, changes made to *new.record* in `BEFORE` triggers are put into the record written to disk. Other changes to *new.record* are ignored. *new.recordID*, *old.recordID*, and *old.record* are read-only variables.

BASIC programs can access attributes in the current record in the usual way, for example, by referring to the field number. For example, if the SALARY attribute were the eighth field in the EMPLOYEES file, the following line of UniVerse BASIC code would test whether SALARY had changed:

```
IF OLD.RECORD<8> # NEW.RECORD<8>...
```

Input of any kind is not allowed in trigger programs. Print output is allowed. However, if the trigger fires as the result of an `SQLExecDirect` or `SQLExecute` function call, no print output is returned.

Opening files

Trigger programs should open all tables and files to a common variable. This avoids having to open and close each table or file whenever the program references it. For example, when you open files to a common variable, a statement that updates many records, each of which fires a trigger that writes an audit record, opens the audit file only once.

Warning: Do not add, drop, enable, or disable any triggers for a table or file while the table or file is in use. To avoid errors, make sure you lock the record prior to applying a WRITE trigger.

index-based subroutines

Index-based subroutines tend to be more lightweight and flexible than conventional triggers. They are useful if you need to audit events that change the database, but you don't need the overhead associated with explicitly creating a trigger through a `CREATE.TRIGGER` command. Instead, you follow a set of guidelines to produce trigger behavior from an index-based subroutine.

The trigger behavior is produced by creating a UniVerse BASIC subroutine, which is then called within an I-descriptor field for a file. You then create an index on that I-descriptor using the NO.NULLS keyword. The index updates when a record on the file is modified via a `DELETE, INSERT`, or `UPDATE` command, which causes the subroutine in the I-descriptor to fire.

The following example creates the INDEX and adds the I-descriptor to the file.

```
>CREATE.FILE TEST.IDX 2 1 1
>ED DICT TEST.IDX INDEX.ITYPE
New record.
----: I
0001= I
0002= SUBR("INDEX.SUB")
0003=
0004=
0005= 10L
0006= S
0007=
Bottom at line 6.
----: FI
>CREATE.INDEX TEST.IDX INDEX.ITYPE NO.NULLS
```

The index on the file is evaluated every time the file is modified, which invokes the `INDEX.SUB` subroutine. Therefore, whenever a record in the file is modified via an `DELETE, INSERT`, or `UPDATE` operation, the indexed subroutine can be used to track database file modifications.

The @IDX.IOTYPE variable is a new @variable that can be integrated in the indexed subroutine to determine the type of database operation that caused the indexed subroutine to fire. The value of the @IDX.IOTYPE variable specifies the type of operation being performed.

The following table describes values associated with the @IDX.IOTYPE.

Value	Description
0	The value returned when @IDX.IOTYPE is used outside the context of an indexed subroutine.
1	The value returned when the SUBR is called because an `INSERT` operation is performed.
2	The value returned when the SUBR is called because a `DELETE` operation is performed.
3	The value returned when the SUBR is called because an `UPDATE` operation is used to evaluate the original value operation.
4	The value returned when a SUBR is called because an `UPDATE` operation is used to evaluate the new value operation.

The @IDX.IOTYPE value is stacked, so that if the SUBR variable is not called in a nested format, the return value will be the current caller's status.

Programming example

The following example demonstrates how to use the @IDX.IOTYPE variable with the supported `INSERT, DELETE, UPDOLD`, and `UPDNEW` values.

```
SUBROUTINE INDEX.SUB(RTNVAL)
COMMON /INDEX.SUB/ OPENFLAG,F.AUDIT,OLDRECORD
RTNVAL = ""
IF NOT(OPENFLAG) THEN
OPEN "AUDIT.FILE" TO F.AUDIT ELSE STOP "CANNOT OPEN AUDIT.FILE"
OPENFLAG = 1
END
*
* The following case statement can be used to execute any specific
* operations related to the type of operation being performed.
*
AUDIT.REC = ''
BEGIN CASE
CASE @IDX.IOTYPE = "1"
CASE @IDX.IOTYPE = "2"
CASE @IDX.IOTYPE = "3"
OLDRECORD = LOWER(@RECORD)
CASE @IDX.IOTYPE = "4"
AUDIT.REC<2> = OLDRECORD
CASE 1
RETURN
END CASE
IF @IDX.IOTYPE # "3" THEN
RECID = @DATE:"*":SYSTEM(12):"*":@ID
AUDIT.REC<1> = @IDX.IOTYPE
WRITE AUDIT.REC ON F.AUDIT,RECID
END
RETURN
END
```

Chapter 8: Peripheral devices

This chapter describes how UniVerse uses printers, tape drives, and terminals, and how the UniVerse spooler works.

Redirecting output

By default, all the output from UniVerse commands and UniVerse BASIC programs is directed to your terminal. You can redirect this output to:

- A printer
- A tape device
- A diskette
- Another terminal
- A disk file

Note: You must define all devices you want to use from UniVerse in the &DEVICE& file. For details on doing this, see *Administering UniVerse*.

UniVerse queues print requests through the UniVerse spooler on UNIX, and the Windows Print Manager on Windows platforms. For more information about using the UniVerse spooler, see Using the UniVerse spooler, on page 132.

For redirecting output, UniVerse provides:

- 256 logical print channels (0 through 255). Print channel 0 is the default printer.
- 8 logical tape channels (0 through 7).

Here is a summary of the steps you follow to direct output to tapes or printers:

1. If required, use the `ASSIGN` command to gain exclusive use of a device.
2. For printers, use the `SETPTR` (UNIX) or `SETPTR` (Windows Platforms) command to set printer parameters such as page length, number of copies, and output destination.
3. Depending on your device, do one of the following:
   - For printers, use the LPTR keyword from RetrieVe or a `PRINT` statement ON from a BASIC program to direct output to the assigned printer.
   - For tape drives, use the MTU keyword from RetrieVe or a UNIT clause from a UniVerse BASIC program to specify the assigned tape drive.
4. Release the assigned device from your exclusive control with the `UNASSIGN` command. When you log out of UniVerse, all assigned devices are automatically released.

The steps are described in more detail in the following sections.

Assigning devices

You need not assign a device unless you want to have exclusive access to it. You should assign a device in the following cases:

- When you want to access tape drives or disk drives

- When you want to print a large file and there is insufficient disk space in the spool queue to hold your print file
- When you have a long print run that you do not want interrupted

You use the ASSIGN command to assign a device to a channel number. The syntax is as follows:

```
ASSIGN device TO LPTR n [ -WAIT ]
ASSIGN device TO MTU n [ MAP mapname ] [ -WAIT ] [ BLK size ]
```

device is the record ID of a record in the &DEVICE& file that defines the device. You can list the records in the &DEVICE& file by entering `LIST &DEVICE&`.

`LPTR` specifies a print channel. *n* is 0 to 255.

`MTU` specifies a tape channel. *n* is 0 to 7.

`-WAIT` tells the processor to wait if the device is assigned to another user. It will be assigned to you when it is free.

mapname specifies the map for the tape device.

`BLK` *size* specifies a block size, in bytes, for tape devices.

In addition, you can use `P.ATT` and `T.ATT`, which assign a printer and a tape drive, respectively.

Unassigning devices

When you finish using a device, you must unassign it before other users can access it. You can do this in three ways:

- By specifying the command `UNASSIGN` device
- By specifying the `P.DET` or `T.DET` command
- By logging out of UniVerse

Setting printer parameters

The 256 print channels all have the same default settings when you first enter UniVerse. You can examine settings for any print channel with the `SETPTR` (UNIX) command or `SETPTR` (Windows platforms). For example, this listing shows the default settings for print channel 155:

```
>SETPTR
155Unit Number : 155
 Page Width    : 132
 Page Depth    : 66
 Top Margin    : 3
 Bottom Margin : 3
 Print mode    : 1 - Spooled Output
```

This is what the parameters mean:

Parameter	Description
Unit number	The number of the print channel. Values are 0 through 255.
Page width	The line length, specified in characters.
Page depth	The page length, specified in lines.
Top margin	The number of blank lines to leave at the top of the page.

Parameter	Description
Bottom margin	The number of blank lines to leave at the bottom of the page.
Print mode	The destination of the print output: 1 = send the output to the spool queue or Print Manager. 2 = send the output to an assigned device. 3 = send the output to a hold file. It is stored as a record in the &HOLD& file in your account. To print a file stored in &HOLD&, use the SPOOL command.

You can customize these settings for up to 16 print channels with the SETPTR command. The new settings remain in effect until you change them or until you log out of UniVerse.

The syntax is as follows:

```
SETPTR [ channel , l.len , p.len , top, bottom, mode, options ]
```

You must specify the parameters in the order shown, separated by commas. The options in the syntax let you specify further parameters including the number of copies to be printed, whether to eject a blank page between print jobs, and so on.

Putting print jobs on hold

Use the SETPTR command to specify two kinds of hold status for a print job sent to the queue:

- Use SETPTR –HOLD to create a temporary hold job that is retained in the queue only until it is released for printing.
- Use SETPTR –RETAIN or SETPTR –REQUEUE to create a permanent hold job that is retained until you explicitly delete it from the queue.

Printing RetrieVe output

You can print RetrieVe output by adding the LPTR keyword to a sentence. The following examples show different ways to customize the output.

Using the default printer

To direct the output of a RetrieVe command to the default printer (print channel 0), add the LPTR keyword to the sentence. For example:

```
>LIST INVOICES WITH AMOUNT > 1000 LPTR
```

Changing print parameters before printing

To print a file using a different line width or a special form:

1. Use the SETPTR command to change the settings of one of the print channels.
2. Use the LPTR keyword to direct output to that print channel.

For example:

```
>SETPTR 4,80,66,,,1,FORM PORTRAIT, COPIES 3
>LIST FORECAST SUMMARY.REPORT LPTR 4
```

The SETPTR command defines the printer settings for print channel 4 to use a line width of 80 characters, a page length of 66, and a form called PORTRAIT, and to print 3 copies. The LIST command then uses that printer definition to print three copies of the report defined by the phrase SUMMARY.REPORT.

Sending output to an assigned printer

To send a report directly to a printer rather than to the print queue, assign the printer to a print channel for your exclusive use:

```
>ASSIGN LASER TO LPTR 3
SETPTR 3,80,66,,,1
LIST INVOICES BY ACCOUNT LPTR 3
UNASSIGN LASER
```

The device defined as LASER in the &DEVICE& file is assigned to print channel 3, and its characteristics are set with the SETPTR command. The LIST command sends its output to LASER through print channel 3. UNASSIGN releases control of LASER so other users can have access to it.

Sending output to a hold file

To direct output to a record in the &HOLD& file, specify mode 3 with the SETPTR command:

```
>SETPTR 2,80,66,,,3
LIST INVOICES BY ACCOUNT LPTR 2
```

Print channel 2 is set to output to the &HOLD& file. The LIST command sends its output to a record in the &HOLD& file as if it had the format set by the SETPTR command.

Printing complete records

Use the SPOOL command to send complete records of UniVerse files to the print queue for printing on the default printer (print channel 0).

Printing to and from tape

You can send records in print format to tape instead of to a printer. To do this, do the following:

On UNIX systems

1. Set up a printer form called TAPE.
2. Use ASSIGN or P.ATT to assign a tape drive to a print channel.
3. Use SPOOL *filenamerecord* -FORM TAPE to spool records to tape.

Use the SP.TAPE command to send records stored on tape in print format to the printer.

Printing UniVerse BASIC output

In a UniVerse BASIC program, you assign printers and change printer settings by invoking the `ASSIGN` and `SETPTR` commands with the `EXECUTE` statement. The UniVerse BASIC `PRINT` statement ON directs output to logical print channels. Here are some lines of a program that assign a printer called LP to print channel 7, set the printer characteristics, then print a string to print channel 7:

```
0429 EXECUTE "ASSIGN LP TO LPTR 7"
0430 EXECUTE "SETPTR 3,80,66,,,1"
0431 PRINT ON 7, "This line is printed on the printer called LP"
```

The UniVerse spooler (UNIX)

Windows servers use only the Windows Print Manager. Therefore, the rest of this chapter applies only to UNIX servers.

The UniVerse spooler queues and manages print requests for UniVerse users on UNIX servers. When you send UniVerse output to the print queue, a print file is created in the UniVerse spooler directory. When the print file is closed, it is available for printing.

The spooler searches for a printer that matches the characteristics needed for the print file. If the spooler finds a printer, it prints the file. If no printer is available, the spooler queues the file and puts it in the wait state.

Before printing, the spooler locks the printer, which prevents any other UNIX process from opening it, and also creates a lock file in the spooler directory. The spooler uses the lock file to coordinate different logical printers that are defined for the same physical printer.

After the print file is printed, the spooler removes the lock file, closes the device, and erases the print file. It then removes the entry for that print job from the spool queue and is ready to schedule another job to that printer.

See *Administering UniVerse* for more information about any of the following:

- Installing, configuring, and removing printers
- Starting, stopping, and restarting printers
- Assigning forms to printers
- Starting, stopping, and restarting the UniVerse spooler

Using the UniVerse spooler

You can use the UniVerse spooler to do any of the following:

- Check the status of printers
- Specify print job destinations
- Set and change print job characteristics
- Check the status of print jobs in the spool queue
- Manage print jobs in spool queues
- Manage hold files

Note: Some printer administration tasks can be performed only by a UniVerse Administrator. Where appropriate, the UniVerse Administrator can define a subset of printers as a separate printer group that can be controlled by other UniVerse users. Printer groups and the users allowed to access them are defined in the `print_group` file, located in the UniVerse spooler directory, usually `/usr/spool/uv`.

You perform these tasks from the **Printer Administration** menu. Display this with the `PRINT.ADMIN` command.

The Printer Administration menu has three options: **Status**, **Control job**, and **Modify job**.

- Use the **Status** option to display a report on the status of printers and their queues.
- Use the **Control job** option to change the status of print jobs.
- Use the **Modify job** option to change print job characteristics.

Checking print job status

When you choose the **Status** option, a submenu provides a variety of possible status reports. The Status submenu provides the following options:

Option	Description
Quick status	Lists all print jobs queued for printing, whether active, waiting, suspended, or in a hold state.
Empty queues too	Lists all currently defined printers and their queues, even if they are empty.
Active jobs	Lists print jobs that are being printed. You can also display print job status with the \ command or from a UNIX shell with the `usa` command.

Changing print job status

Choose **Control job** to make changes to the print jobs themselves—for example, cancel a print job, or print it again.

The **Control job** submenu provides the following options:

Option	Description
Kill a job	Removes a job from the queue, whether active, waiting, on hold, or suspended.
Suspend	Suspends printing of an active print job, retaining the print file in the queue. This option can be used only by users with system administration privileges or users whose names are included in the `print_group` file.
Continue	Resumes printing of a suspended print job at the point where it left off printing. This option can be used only by users with system administration privileges or users whose names are included in the `print_group` file.
Hold	Changes status of print job to “hold.” Such a job is held on the queue only until it is released for printing. Use **Modify job** if you want to give a print job permanent hold status.
Release	Releases a held print job for printing. The print file is not retained in the queue after printing.

Option	Description
Print again	Prints another copy of the print job. The job is retained in the queue after it is printed.

Modifying print job characteristics

Choose **Modify job** to change print job characteristics, such as the number of pages to print, which pages to print, and when to print.

The **Modify Job** submenu provides the following options:

Option	Description
Number of copies	Changes the number of copies to print.
Pages to print	Specifies a range of pages to print.
Lines to print	Specifies a range of lines to print.
Hold after printing	Changes status of print job to “hold.” Such a job is held on the queue even after it is printed.
Destination printer	Changes the printer specified to print the job.
Schedule priority	Changes the assigned priority of the job. The highest priority is 1, the lowest is 255.
Form	Changes the form specified for the print job.
Time delay	Changes the relative or absolute time delay before the job is printed. You can specify either the relative time you want the spooler to print the job (for example, 4 hours from now) or the absolute time (for example, 3:30 p.m.).

Using tape drives

You must always use the `ASSIGN` or the `T.ATT` command to assign a tape drive to a tape channel before using it. When you complete the tape operation, use the `UNASSIGN` or the `T.DET` command to release the tape drive from your exclusive control.

Copying records from disk to tape

Use the `T.DUMP` command to copy file records from disk to tape. The simplest form of the command is as follows:

```
T.DUMP filename
```

With this syntax, `T.DUMP` copies all records in the data file from disk to tape using the default tape channel 0. `T.DUMP` puts an end-of-file mark on the tape at the end of each operation. This mark separates the files on the tape.

Copying records from tape to disk

Use the `T.LOAD` command to copy tape files created with `T.DUMP` back to disk. The simplest form of the command is as follows:

```
T.LOAD filename
```

When you use this sentence, `T.LOAD` copies all the records in the tape file assigned to magnetic tape channel 0 to the data file on disk. A number of commands are used to advance or rewind the tape to the correct position. These are summarized in the following table.

You can use a RetrieVe selection expression to copy selected records, but you cannot specify a sort expression. You can overwrite records that already exist on the disk by specifying the OVERWRITING keyword.

Name	Description
`ASSIGN`	Assigns control of the tape drive to you.
`SP.TAPE`	Prints a print job stored on tape.
`T.ATT`	Attaches the tape drive to your account.
`T.BCK [`*`nn`*`]`	Backs up a specified number of tape records, or to the end-of-file mark.
`T.DET`	Detaches the tape drive from your account.
`T.DUMP`	Writes selected records from disk to tape.
`T.EOD`	Advances a tape to the end-of-data mark.
`T.FWD [ `*`nn`*` ]`	Advances a specified number of tape records, or to the end-of-file mark.
`T.LOAD`	Loads selected records from tape to disk.
`T.RDLBL`	Reads a tape label at the beginning of the reel.
`T.READ`	Reads the next tape record, then displays it. Use the `LPTR` keyword to print the tape record.
`T.REW`	Rewinds the tape to the load point.
`T.SPACE [ `*`n`*` ]`	Advances a specified number of files on the tape, or to the end-of-file mark.
`T.UNLOAD`	Rewinds the tape to the load point and then unloads the tape.
`T.WEOF`	Writes an end-of-file mark on the tape.
`T.WTLBL`	Writes a tape label at the current position of the tape.
`UNASSIGN`	Relinquishes control of the tape drive.

Terminals

UniVerse needs to know the type of terminal you are using in order to format output correctly for menus, reports, and so on. UniVerse stores terminal definitions in its own terminfo database in the UV account directory.

A UniVerse Administrator can modify these terminal definitions (described in detail in *Administering UniVerse*), and other UniVerse users can modify their own terminal settings.

Displaying your terminal setting

Use the `GET.TERM.TYPE` command to display your current terminal setting, including the code, model, and a brief description.

Setting terminal characteristics

Use `SET.TERM.TYPE` to tell the system the type of terminal you are using. The syntax is as follows:

`SET.TERM.TYPE [ code ] [ options ]`

code is the terminal type code, which is the name of the file containing the terminal definition in the terminfo database.

options is one or more of the following:

Option	Description
AUTONL	Generates a new line sequence if a line longer than the width of the screen is typed.
FUNDAMENTAL	Loads the default key bindings.
HUSH	Suppresses terminal output.
LENGTH *n*	Sets the length of the terminal to *n* lines.
NEEDNL	Lets UniVerse generate new line sequences.
VERIFY.SUP	Means that no prompting is required if an invalid terminal type is entered.
WIDTH *n*	Sets the width of the terminal screen to *n* characters.

If you do not specify a code, UniVerse prompts you for one. Enter a question mark (**?**) at the prompt to list supported codes.

You can also use the `TERM`, `PTERM` (UNIX) , or `PTERM` (Windows platforms) command to set or display printer and terminal characteristics.

Chapter 9: Using ReVise

This chapter describes ReVise, a UniVerse processor that lets you add or change records in dictionaries and data files.

What is ReVise?

ReVise is an interactive processor that can be used to create records in a data file or entries in a file dictionary. You can also use ReVise to add, change, or delete data in these records. To invoke ReVise, use the following simplified syntax:

```
REVISE [ DICT ] [ filename ] fields ]
```

DICT specifies the file dictionary. *filename* is the name of the file containing the records you want to revise.

fields are the names of one or more fields whose data you want to revise. If you specify *fields*, ReVise prompts only for those fields. If you do not specify *fields*, ReVise prompts for the fields defined in a phrase called @REVISE, if it exists in the file dictionary. If there is no @REVISE phrase, ReVise uses the fields defined in the @ phrase, also in the file dictionary. If neither an @REVISE phrase nor an @ phrase exists, ReVise creates an @REVISE phrase containing all the data descriptor fields in the dictionary and uses those fields.

fields not named in the @REVISE phrase or the @ phrase are not displayed. Therefore, you can restrict user access to certain fields in a record by excluding the names of those fields that you want to protect from the @REVISE phrase or the @ phrase.

Note: When ReVise displays field name prompts, it numbers field 0 (the record ID in the file dictionary) as 1. The other numbers in the display do not correspond to the actual field numbers as they exist in the data file.

ReVise uses either the @REVISE phrase or the @ phrase to determine which field names to use as prompts and in what order they should appear. The following example shows that fields the user can enter are numbered in the order in which they appear in the @REVISE phrase. The record ID (the order number) is the first field in the @ REVISE phrase, the date is the second field, the customer number the third field, and so on.

```
>LIST DICT ORDERS
DICT ORDERS 03:40:17pm  11 Jan 1995 Page 1
                     Type &
Field..... Field. Field........
Name...... Number Definition...
@ID          D   0
ORDER.NO       D   0
DATE           D   1      D2
CUST.NO         D   2
PROD.NO            D   3
QTY          D   4   MD
ITEM.TOTAL   I           QTY * SELL  MD2$,
SELL           I             TRANS(INVENTO     MD2$,
                                   RY,PROD.NO,SE
                                   LL,'X')
ORDER.TOTAL      I           SUM(ITEM.TOTA   MD2$,
                                   L)
@            PH      ORDER.NO
                                   CUST.NO
                                   PROD.NO QTY
```

```
                                          ORDER.TOTAL
                                          ID.SUP
BOUGHT       PH       PROD.NO QTY
@REVISE           PH       ORDER.NO DATE
                                          CUST.NO
                                          PROD.NO QTY
12 records listed.
>REVISE ORDERS
ORDERS
REVISE.1 Tue Dec 06 14:30:25 1994
RECORD ID= 999
ORDERS -Screen 1-FIRST SCREEN Mon Dec 05 14:50:51 1994
1 RECORD ID 999
2 DATE Red Silicon Ball
3 CUST.NO 45
```

Specifying ReVise prompts

If you do not enter a file name, ReVise prompts you to enter a process name. The process name determines the type of file and the set of prompts ReVise will use for entering data. You can specify one of the following six processes:

Process	Description
ENTER.DICT	Creates an entry in a file dictionary. ReVise prompts you for the name of the file dictionary. For example, if you enter the file name DICT CUSTOMERS, ReVise prompts you with field names for an entry in the file DICT CUSTOMERS.
ENTER.DICTAS	Creates A-descriptor and S-descriptor entries in a UniVerse file dictionary (not in a Pick-style dictionary). Use ENTER.DICTAS to create a 10-line dictionary entry with a TYPE code of A or S. ReVise prompts you for the name of the file dictionary, as it does when you use ENTER.DICT.
ENTER.MENUS	Creates a menu in a menu file. ReVise prompts you for the name of the menu file. For example, if you enter the file name MY.MENUS, ReVise prompts you with field names for an unformatted menu definition record.
ENTER.FMENUS	Creates a formatted menu in a menu file. ReVise prompts you for the name of the menu file.
ENTER.S	Creates a stored sentence in the VOC file.
ENTER.SELECTOR	Creates a menu selector record in the VOC file. Selector records point to menus stored in menu files. Entering the name of the selector at the system prompt invokes the menu.

When you revise records in a data file, you are first prompted to enter a record ID. Note that ReVise does not allow empty record IDs, that is, IDs of zero length that are known to have no value. ReVise, like the Editor, lets you enter special characters using control characters. The prompt is the column heading defined in the @ID dictionary entry—usually the name of the data file itself.

```
>REVISE ORDERS
ORDERS REVISE.1 Tue Dec 06 12:40:10 1994
RECORD ID=
```

If you enter an existing record ID, the current values of the record are displayed, and you are prompted to change them. Note that if you enter an invalid record ID for a distributed or part file at the change

prompt, ReVise may be unable to locate the record. You will need to reenter the original valid record ID in field 1.

```
ORDERS -Screen 1-FIRST SCREEN Tue Dec 06 12:32:19 1994
1 RECORD ID     102
2 DATE          Red Silicon Ball
3 CUST.NO       45
S1 == FIRST SCREEN
S2 == PROD.NO QTY
CHANGE=
```

If you enter a new record ID, the message New Record appears, then the input prompts for the specified fields are displayed in sequence:

```
ORDERS REVISE.1 Tue Dec 06 12:45:17 1994
RECORD ID= 987
New Record
Order Date=
```

After you have finished entering data, the values you entered are displayed, and you are prompted to change them if you want to:

```
ORDERS -Screen 1-FIRST SCREEN Tue Dec 06 14:37:29 1994
1 RECORD ID     987
2 DATE          23 DEC 94
3 CUST.NO       97

S1 == FIRST SCREEN
S2 == PROD.NO QTY
CHANGE=
```

When you revise entries in a file dictionary, the processing is the same as when you enter data in records in a data file, except that the input prompts are those defined in the @REVISE phrase located in the file DICT.DICT, the metadictionary used by all UniVerse dictionaries. These input prompts are only column headings in a report. Otherwise, they are display or input names for fields or prompts. You can display the contents of DICT.DICT by entering:

```
>LIST DICT.DICT
```

When entering new data in a file dictionary, you are prompted for each field in turn. Enter the record ID at the FIELD= prompt. The same prompts are displayed for data descriptors, I-descriptors, phrases, and X-descriptors.

Entering data on different screens

ReVise uses two different kinds of screen to organize your data into pages of information. All singlevalued fields are prompted for on the same screen, Screen 1. The previous example shows Screen 1 (FIRST SCREEN, S1).

For multivalued fields in a record, ReVise automatically creates additional pages or screens of information, one for each association of multivalued fields and one for each multivalued field that is not part of an association. These screens are called Screen 2 (S2), Screen 3 (S3), and so on. To see Screen 2, you first invoke ReVise for the file ORDERS for record ID (Order No) 102:

```
>REVISE ORDERS
ORDERS REVISE.1 Fri Dec 16 09:37:50 1994
Order No= 102
```

Screen 1 appears and you are prompted by the CHANGE= prompt. You can display screen 2 by entering S2:

```
ORDERS -Screen 1-FIRST SCREEN Fri Dec 16 09:38:34 1994
1 ORDER.NO      102
2 DATE          Red Silicon Ball
3 CUST.NO       Red Silicon Ball
S1 == FIRST SCREEN
S2 == PROD.NO QTY
CHANGE= S2
```

Screen 2 (S2) displays the associated multivalued fields. After you finish entering new data in the multivalued screen, you can enter the item number (1 or 2 from the next example) or respond as described in Responding to multivalued field prompts, on page 141.

```
ORDERS -Screen 2-PROD.NO QTY Fri Dec 16 09:13:03 1994
RECORD ID==> 102
No. ...PROD.NO .QTY
    1               45  1400
    2
Change which line item=
```

Responses to ReVise prompts

The following sections describe various general responses that you can enter at different ReVise prompts.

Responding to the record ID prompt

When you invoke ReVise, the first prompt is for the record ID. You can enter either a record ID or any of the following:

Value	Description
?	Displays help about the record ID prompt.
??	Displays additional help for the record ID prompt. This works only if a user-defined Help file has been set up that contains the additional help messages.
^	Displays Screen 1 and repeats the record ID prompt.
^^	Same as ^.
Q or QUIT	Exits ReVise. If the file dictionary has an &NEXT.AVAILABLE& entry, pressing Q , QUIT , or Enter enters the next available record ID.
Return	Same as Q.

Responding to singlevalued field prompts

When a prompt for a singlevalued field appears, you can enter a value. You can also enter any of the following:

Value	Description
?	Displays help about the field prompt.

Value	Description
??	Displays additional help for the field prompt. This works only if a user-defined Help file has been set up which contains the additional help messages.
TOP	If you are adding new data, TOP exits the record without saving any of the changes you entered. If you are changing existing data, TOP exits the record leaving the data as it was before you changed it. ReVise then prompts for another record ID.
.	Skips to the next required prompt (or the next screen, if it exists).
^	Backs up and redisplays the previous prompt.
^^	Repaints the screen and redisplays the current prompt.
C/old/new/	Changes the old string to the new string.

Responding to multivalued field prompts

ReVise uses separate screens for entering data in multivalued fields. Each multivalued field has its own screen, unless it is part of an association. ReVise displays all associated multivalued fields on the same screen. PROD.NO and QTY are associated multivalued fields in this example:

```
ORDERS -Screen 2-PROD.NO QTY Fri Dec 16 10:06:07 1994
ORDER.NO==> 102
No. ...PROD.NO .QTY
    1               45  1400
    2
Change which line item=
```

If you are adding new data to a set of associated multivalued fields, the first field prompts you for a value, then the second field prompts you, and so on, through the first set of associated field values. Then ReVise prompts for the second set of associated values field by field, and so on.

```
ORDERS -Screen 3-PROD.DESC Fri Dec 16 11:55:36 1994
ORDER.NO==> 888
No. PROD.DESC......
    1
Product Desc=
```

If you are changing data in multivalued fields, the current values are displayed and you are prompted to enter the line number of the item whose data you want to change. After you enter a number, the appropriate field names prompt you to enter your changes.

At any of these field prompts you can enter data or any of the following responses:

Value	Description
?	Displays help about the field prompt.
??	Displays additional help for the field prompt. This works only if a user-defined Help file has been set up which contains the additional help messages.
##	Deletes the entire line item (that is, the entire set of associated multivalues, not just the one in the specified field).
>	Inserts a new line item above the current line item. ReVise displays an explanation and prompts you to enter the new set of associated multivalues.

Value	Description
>#*n*	Copies all associated multivalues of line item *n* and puts it above the current line item.
"	Copies the field value from the preceding line, if one exists.

Responding to the change which line item= Prompt

After you have finished entering new data using a multivalued screen, or when you display data from an existing record, ReVise displays the Change which line item= prompt. Enter a line item number or any of the following responses:

Value	Description
?	Displays help about the Change which line `item=` prompt.
??	Lists the possible responses to the currently activated ReVise prompts.
TOP	Exits the multivalued screen and returns to Screen 1, saving all changes made to the multivalued data.
^	Repeats the Change which line `item=` prompt.
^^	Repaints the screen and redisplays the Change which line `item=` prompt.
.	Exits the current record saving all changes, and prompts for another record ID.
Return	ReVise displays one of the following: The next page of information if there are more values than fit on a screen The next screen of multivalued fields if you are entering data The first screen containing the singlevalued fields if neither of the preceding conditions is true

Responding to the CHANGE= Prompt

After you have finished entering data in a new record, or when you enter the name of an existing record at the record ID prompt, ReVise displays the CHANGE= prompt. You can now enter any of the following responses:

Value	Description
?	Displays help about the CHANGE= prompt.
??	Displays a list of all possible responses to the currently activated ReVise prompts.
TOP	If you are adding new data, TOP exits the record without saving any of the changes you entered. If you are changing existing data, TOP exits the record leaving the data as it was before you changed it.
^	Repeats the CHANGE= prompt.
^^	Repaints the screen and redisplays the CHANGE= prompt.

Value	Description
.	Saves any changes you have made, exits the current record, and prompts for another record ID.
Return	Same as . .
DELETE	Deletes the current record and prompts for another record ID.
Q or QUIT	If you are using a select list to specify which records you want to add or change, Q exits the current record without saving any changes, clears the select list, and prompts for another record ID.
S *n*	Displays screen *n* for processing. The screen must be one of those listed on Screen 1.

Changing file dictionaries

You can use ReVise to add a new dictionary entry or to change the contents of an existing dictionary entry. If you want to add or change D-descriptor or I-descriptor entries, use the following ReVise syntax:

```
REVISE DICT filename
```

If you want to add or change A-descriptor or S-descriptor entries, enter:

```
>REVISE
```

At the **Enter Process** name prompt, use the keyword ENTER.DICTAS:

```
Enter Process name ENTER.DICTAS
Enter file name
```

Here is an example of how ReVise displays the contents of a dictionary entry:

```
>REVISE DICT SUN.MEMBER
RETRIEVE DICTIONARY DEFINITION REVISE.1  Fri Dec 16 08:51:08 1994
FIELD= FNAME
RETRIEVE DICTIONARY DEFINITION -Screen 1- Fri Dec 16 08:51:19 1994
1  FIELD     FNAME
2  TYPE      D
3  LOC       2
4  CONV
5  NAME      FIRST NAME
6  FORMAT    5T
7  S/M       S
8  ASSOC
9  SQLTYPE
CHANGE= <Return>
```

To change a value, enter the appropriate field number. For example, if you want to change the value in the field NAME, enter 5 . You are prompted to enter a new name. If you do not want to change any of the values, press Enter.

If you enter the record ID of a new dictionary entry at the FIELD= prompt, you are asked to enter data for each field of the new entry. ReVise displays the following prompts:

Prompt	Your response
New Record	This message appears when the name entered at the FIELD= prompt does not exist in the dictionary.

Prompt	Your response
TYPE+DESC	Enter one of the following types. A description of the item is optional. D – for a data descriptor. I – for an I-descriptor. PH – for a phrase. X – for a user record.
LOC	D - Enter the relative location of the descriptor in the file. If this is the first field, enter 1; if the second, enter 2; and so on. The record ID is always assigned location 0. I – Enter the expression that produces the value for this field. PH – Enter the field names, keywords, and selection and sort expressions that make up the phrase. X – Enter whatever data is to make up the user record.
CONV=	Enter a conversion code. This entry is optional. Press Enter to skip this specification. If you are creating a phrase or an X-descriptor and you are finished adding data, type a period (.) at this prompt to save your changes and display the CHANGE= prompt, skipping all remaining fields.
DISPLAY NAME=	Enter the name you want for the field identifier. This name is also used as an input prompt in a RetrieVe report or prompts. This optional field can include spaces. If you do not enter a name at this prompt, UniVerse uses the record ID of the dictionary entry. Press Enter to skip this specification.
FORMAT=	Enter the output format. For data descriptors and I-descriptors, you must enter the width of the field and the justification for the text (R and L for right and left justify, and T for text). You can also enter a field background character, an editing code, and a mask field. The syntax of this entry is: *width* [*background*] *justification* [*edit*] [*mask*] Press Enter to skip this specification.
S/M=	Enter S or M to specify whether the field or I-descriptor contains single or multiple values. Enter S if this field contains only one value, enter M if it can contain one or more values. S is assumed by default. Press Enter to skip this specification.
ASSOC=	For a multivalued field that is to be associated with other multivalued fields, enter the record ID of the phrase containing the names of the associated fields. If a multivalued field is not associated with another, this field can be left blank. Press Enter to skip this specification.
SQL DATA TYPE=	(Optional) Enter the SQL data type of this field, used by SQL SELECT statements. Press Enter to skip this specification.

After you have entered all the values for an entry, ReVise displays the values and prompts you to change them. When the values are as you want them, press Enter to save the entry to the dictionary and return to the FIELD= prompt.

This series of prompts is repeated for every record that you enter into the dictionary. When you finish defining records in the dictionary, press Enter or enter QUIT (or Q) at the FIELD= prompt. This returns you to the UniVerse prompt.

Changing data files

You must define the entries in a file dictionary before you use ReVise to add or change data in the data file. If you do not, ReVise prompts only for the record ID. To add or change data in the data file, use the following ReVise syntax:

```
REVISE filename [ fields ]
```

filename is the name of the file to which you are adding data or whose data you are changing.

fields are the names of one or more fields whose data is to be added to or changed.

ReVise uses separate screens for singlevalued and multivalued fields. The screens are numbered sequentially. Screen 1 contains all the singlevalued fields. Screen 2, Screen 3, and so on, (depending on the number of associated and unassociated multivalued fields in the file), contain multivalued fields.

Singlevalued fields

When you first invoke ReVise, Screen 1 appears. ReVise always displays record ID first.

```
>REVISE SUN.MEMBER
SUN.MEMBER REVISE.1 Fri Dec 16 08:56:28 1994
MEMBER ID= 9875
```

In the example, the record ID prompt is `MEMBER ID=`. To begin adding or changing data to the single-valued fields, enter a value for the record ID.

ReVise prompts for the next single-valued field and waits for your response. To leave a field blank, press Return.

```
FIRST NAME= BARRY
```

ReVise continues displaying the prompts for singlevalued fields on Screen 1 until you have entered data or skipped past all fields. If the file contains only singlevalued fields, ReVise next displays the CHANGE= prompt. For multivalued fields, ReVise displays Screen 2, and so on, before the CHANGE= prompt. The next example shows how to use Screen 1 to create a new record with single-valued fields (the data is entered for each field):

```
MEMBER ID= 9876
New Record
FIRST NAME= BARRY
LAST NAME= SMITH
STREET ADDRESS= 30 MAIN ST.
CITY= BOSTON
STATE= MA
ZIP= 02166
YEAR JOINED= 1989
CHANGE=
```

After you have finished with one record, ReVise prompts for another record ID and repeats the sequence:

```
MEMBER ID= 9877
```

```
New Record
FIRST NAME= JOHN
```

Unassociated multivalued fields

Each unassociated multivalued field is displayed on a separate screen. The multivalued screens are numbered 2, 3, 4, and so on. The following example has one multivalued field (Screen 2):

```
>REVISE SUN.MEMBER
SUN.MEMBER REVISE.1 Fri Dec 16 09:02:14 1994
MEMBER ID= 4500
SUN.MEMBER -Screen 1-FIRST SCREEN Fri Dec 16 09:02:21 1994
1 RECORD ID 4500
2 FNAME DAVID
3 LNAME DOUGHERTY
4 STREET 5 DENBY STREET
5 CITY WALTHAM
6 STATE MA
7 ZIP 02154
8 YR.JOIN 1988
S1 = FIRST SCREENS
2 = INTERESTS LEVEL
CHANGE= S2
SUN.MEMBER -Screen 2-INTERESTS Fri Dec 16 09:02:51 1994
RECORD ID ==> 4500
No. INTERESTS
1 FISHING
2 BASKETBALL
3
Change which line item=
```

ReVise prompts for values for the multivalued field until you press Enter. If there are more multivalue screens, ReVise displays the next screen.

After you have entered information in all the multivalued fields, ReVise displays the new values and prompts you to change them. When the data is as you want it, press Enter at the Change which line item= prompt. ReVise returns to Screen 1 and displays the CHANGE= prompt. You can change any information in the record before you go on to the next record. Enter `??` to get a list of the possible responses to CHANGE=.

Enter the line number of the field that you want to change, or enter `S` *n*, where *n* is the screen number containing the multivalued field that you want to change. If you enter a screen number, ReVise displays the multivalued data and the prompt Change which line item=. ReVise assigns a number to each line item on the screen. Enter the number of the line item you want to change.

```
>REVISE SUN.MEMBER
SUN.MEMBER REVISE.1 Fri Dec 16 09:02:14 1994
MEMBER ID= 4500
SUN.MEMBER -Screen 1-FIRST SCREEN Fri Dec 16 09:02:21 1994
1 RECORD ID 4500
2 FNAME DAVID
3 LNAME DOUGHERTY
4 STREET 5 DENBY STREET
5 CITY WALTHAM
6 STATE MA
7 ZIP 02154
8 YR.JOIN 1988
S1 = FIRST SCREENS
```

```
2 = INTERESTS LEVEL
CHANGE= S2
SUN.MEMBER -Screen 2-INTERESTS Fri Dec 16 09:02:51 1994
RECORD ID ==> 4500
No. INTERESTS
1 FISHING
2 BASKETBALL
3
Change which line item=
```

When the data is as you want it, press Enter. ReVise prompts for another record ID and repeats the sequence beginning again at Screen 1.

Associated multivalued fields

If a multivalued field is associated with other fields, the associated fields are prompted for on one screen.

In the following example, values in the INTERESTS field are associated with values in the LEVEL field.

```
SUN.MEMBER -Screen 2-INTERESTS LEVEL Fri Dec 16 09:07:27 1994
RECORD ID ==> 4500
No. INTERESTS              LEVEL
1   SAILING                I
2   FISHING                II
3   ARCHERY                I
4
INTERESTS= SWIMMING
LEVEL=
```

ReVise prompts for each field in the association. The interest SWIMMING must be associated with a level in the example above. When all field prompts have been responded to, the prompt sequence is repeated for the next line item. Enter a value for each field and press Enter. ReVise displays a field that is part of an association, but if it is not one of the fields you specified explicitly, or if it is not in the @REVISE or @ phrase in the dictionary, the field cannot be updated.

ReVise with an active select list

You can use a select list to revise records. If select list 0 is active, ReVise displays each record specified by the select list in order instead of prompting for a record ID. After you finish making changes to the current record, press Enter to save the changes. The next record from the list is displayed. After all records in the select list have been processed, ReVise prompts for a record ID, just as it does when you are modifying records without a select list.

To use a select list with ReVise, you must first use one of the commands that creates or activates a select list (such as `SELECT` or `GET.LIST`), then enter the `REVISE` command at the next prompt (>>).

ReVise can use only select list 0. Here is an example of a select list change screen:

```
>SSELECT SUN.MEMBER
15 record(s) selected to SELECT List #0

>>REVISE SUN.MEMBER
SUN.MEMBER -Screen 1-FIRST SCREEN Fri Dec 16 09:17:03 1994
1 RECORD ID  2342
2 FNAME         BARRY
3 LNAME         ADDAMS
```

```
4 STREET        420 CENTRE STREET
5 CITY      NEWTON
6 STATE         MA
7 ZIP           02159
8 YR.JOIN   1984
S1 == FIRST SCREEN
S2 == INTERESTS LEVEL
CHANGE= <Enter>

SUN.MEMBER -Screen 1-FIRST SCREEN Fri Dec 16 09:17:25 1994
1 RECORD ID     4102
2 FNAME             LESLIE
3 LNAME             BROWN
4 STREET            321 RODEO ROAD
5 CITY          RED ARROW
6 STATE             OK
7 ZIP               47487
8 YR.JOIN       1985
S1 == FIRST SCREEN
S2 == INTERESTS LEVEL
CHANGE= QUIT
SUN.MEMBER REVISE.1 Fri Dec 16 09:17:37 1994
SUN.MEMBER= <Enter>
>
```

Chapter 10: Menus

This chapter describes how to use the menu processor to create and use menus. Although menus can also be created with the UniVerse Editor, the menu processor is much easier to use.

Menus in UniVerse

UniVerse lets you store sentences and paragraphs to eliminate repetitive typing, save time, and automate your application. You can further automate your application by creating menus. A menu contains a list of actions to be performed. You can list the actions by number or display them in a Motif-style pull-down menu.

To select an action, either enter the number or press the Enter key or mnemonic character on a Motif menu. The actions are names or descriptions of sentences that are stored as part of the menu. Unlike when you use stored sentences, you do not have to remember many sentence names to use options on a menu. You have only to remember one menu name, examine its choices, and then choose the appropriate one for the task you want to perform.

The menu processor is itself menu-driven. Use the **Menu Maintenance** menu to create, change, print, and delete menus. The menu processor uses ReVise for data entry. For information about valid ReVise responses, see Using ReVise, on page 137.

File dictionary for menu files

UniVerse has a master dictionary for all menu files. This is the file dictionary of the UNIVERSE.MENU.FILE located in the UV account. For the options of the Menu Maintenance menu to work, you must use this dictionary for your menu file.

To make the UNIVERSE.MENU.FILE dictionary the dictionary for any menu file, first create the data file using `CREATE.FILE`, then use the Editor to change field 3 to point to the dictionary of UNIVERSE.MENU.FILE, as follows:

```
>CREATE.FILE DATA
MY.MENUS 3 1 2 MENU FILE IN MY ACCOUNT
Creating file "MY.MENUS" as Type 3,
Modulo 1, Separation 2.
 >ED VOC MY.MENUS
3 lines long


----: P
0001: F MENU FILE IN MY ACCOUNT
0002: MY.MENUS
0003:
Bottom at line 3.
----: LOAD UNIVERSE.MENU.FILE
Starting line/field number - 3
Ending      line/field number - 3
1 lines/fields loaded.
0003: /u1/uv/D_MENU.FILE
Bottom at line 4.
----: FI
"MY.MENUS" filed in file "VOC."
```

Menu Maintenance menu

To invoke the **Menu Maintenance** menu, enter `MENUS` at the UniVerse prompt:

```
UniVerse Menu Maintenance
    1. Enter/Modify      a menu
    2. Enter/Modify      a formatted menu
    3. Display           a summary of all menus on a menu file
    4. Display           the contents of a menu
    5. Enter/Modify      a VOC menu selector item
    6. Enter/Modify      a VOC stored sentence item
    7. Display           all menu selector items on the VOC file
    8. Display           all stored sentence items on the VOC file
9. Display              the dictionary of a file
10. Print               a summary of all menus on a menu file
11. Print               the contents of a menu
12. Print               a virtual image of a menu on the printer
13. Print               the contents of a formatted menu
14. Print               the dictionary of a file
15. Print               detail of a menu, including VOC records
                        referenced
Which would you like? ( 1 - 15 ) ?
```

To create a formatted menu, choose option **2** from the **Menu Maintenance** menu. UniVerse asks for the name of the menu file and the name of the menu. If the menu is new, the New Record message appears, and you see a new screen. The new screen is a menu definition screen. Use ReVise to enter a description of each option of the menu. The data you enter becomes the menu definition record that is stored in the menu file.

The menu definition record fields and their definitions are as follows:

Field	Name	Definition
1	TITLE	Menu title.
2	DESC	A multivalued field containing the text of the menu options. This text describes the action that occurs when the option is chosen.
3	ACTION	A multivalued field containing the name of a command that is executed when the option is chosen.
4	EXP	A multivalued field containing a brief explanation of what the option does. This field provides an additional help message when the user enters the option number followed by a ?.
5	QUERY	Prompt to replace the default, Which would you like? (1-n)=.
6	EXITS	Commands to exit the current menu.
7	STOPS	Commands to exit the menu system and return to the calling process.

Field 1 contains the title of the menu to be displayed on the screen.

Fields 2, 3, and 4 are associated multivalued fields. Field 2 contains descriptions for each numbered option on the menu. In a formatted menu you can specify where each option is to be displayed on the terminal. Field 3 contains the record IDs of each UniVerse command, stored sentence, paragraph, proc, BASIC program, or menu invoked by each option on the menu. One action must be specified for

each description in field 2. The actions are specified in the same order as the descriptions. Field 4 can contain optional explanations of the menu items.

When you use option 1 from the **Menu Maintenance** menu, `DESC, ACTION, and EXP` are displayed in order for each option. Menus created using option 1 use the default values. Fields 5 through 7 are not displayed. The default value for QUERY (field 5) is as follows:

```
Which would you like (1-n) =
```

The default value for `EXITS` (field 6) is Enter. There is no default value for `STOPS` (field 7).

If you want to change the default values in fields 5 through 7, use option 2 from the **Menu Maintenance** menu.

The following example shows the sequence of prompts and the output that results when you select option 1 from the **Menu Maintenance** menu:

```
NAME OF MENU FILE= MENU.FILE
MENU PROCESS DEFINITIONS REVISE.1 Mon Dec 05 10:54:27 1994
MENU.NAME= MENU1
MENU PROCESS DEFINITIONS-Screen 1-MENU NAME AND TITLE Mon
Dec05 10:54:31 1994
1 MENU.NAME MENU1
2 TITLE PAYROLL MENU
S1 == MENU NAME AND TITLE
S2 == MENU SELECTION ITEMS
CHANGE= S2

MENU PROCESS DEFINITIONS -Screen 2-MENU SELECTION ITEMS Mon Dec 05
10:54:35 1994
MENU.NAME ==> MENU1
No. DESC ACTION EXP
1 ENTER EMPLOYEE INFORMATION EMPLOYEE.UPDATE USE THIS OPTION
TO ENT
2 PROCESS PAYROLL CHECKS UPDATE.CHECKS USE THIS OPTION
TO PRO
3 PRINT PAYROLL CHECKS PRINT.CHECKS USE THIS OPTION
TO PRI
4
Change which line item= <Enter>

MENU PROCESS DEFINITIONS -Screen 1-MENU NAME AND TITLE Mon Dec 05
10:56:14 1994
1 MENU.NAME MENU1
2 TITLE PAYROLL MENU
S1 == MENU NAME AND TITLE
S2 == MENU SELECTION ITEMS
CHANGE= <Enter>
MENU PROCESS DEFINITIONS REVISE.1 Mon Dec 05 10:56:25 1994
MENU.NAME= <Enter>
```

Creating the VOC Menu entry

To invoke the menu from the system prompt, you must create a menu entry in the VOC file. Choose option **5** from the **Menu Maintenance** menu and respond to the ReVise prompts as follows:

Prompt	Response
`SELECTORNAME=`	Enter the record ID of the menu entry. Use this record ID to call the menu.

Prompt	Response
TYPE(M)+DESC=	Enter M to identify this entry as a menu. If you want, include a description of the menu.
MENU.FILE=	Enter the name of the file where the menu is stored.
MENUITEM=	Enter the record ID of the menu in the file.

Here is an example of the PAYROLL menu file:

```
VOC SELECTOR ENTRY REVISE.1 Mon Dec 05 11:00:22 1994
SELECTOR NAME= PAYROLL.MENU
New Record
TYPE(M)+DESC= M PAYROLL DESCRIPTION
MENU.FILE= MENU.FILE
MENUITEM= MENU1
VOC SELECTOR ENTRY -Screen 1- Mon Dec 05 11:00:20 1994
1 KEY PAYROLL.MENU
2 F1 M PAYROLL DESCRIPTION
3 F2 MENU.FILE
4 F3 MENU1
CHANGE= <Return>
VOC SELECTOR ENTRY REVISE.1 Mon Dec 05 11:00:49 1994
SELECTOR NAME= <Enter>
```

Invoking menus

Once the menu entry is created in the VOC file, you can invoke the menu by entering the selector name at the UniVerse prompt. To invoke the menu using the menu entry shown in the previous example, enter the following:

```
>PAYROLL.MENU
```

Creating formatted menus

Formatted menus let you put menu options anywhere on the terminal screen. (Unformatted menus use a default format.) To create a formatted menu, select option 2 from the **Menu Maintenance** menu. You can specify where you want each option and prompt to display on the terminal screen. You can also define a prompt message and change the default exit and stop functions. To position the menu options, specify the column and row coordinates using the following syntax:

@(column,row)option.text

For example, to put an option in the center of the screen, enter the following:

```
@(29,12) SUN.MEMBER INFORMATION
```

Here is a sample menu definition record containing coordinates, customized prompts, and STOP and QUIT messages:

```
NAME OF MENU FILE= MENU.FILE
FORMATTED MENU PROCDEFS REVISE.1 Mon Dec 05 11:32:34 1994
MENU.NAME= MENU2
FORMATTED MENU PROCDEFS -Screen 1-MENU NAME AND TITLE Mon Dec 05 11:32:37
1994
1 MENU.NAME MENU2
2 TITLE SUN.MEMBER MENU
3 QUERY ENTER SELECTION NUMBER
```

```
4 EXITS EXIT,QUIT,Q
5 STOPS STOP,S,HALT
S1 == MENU NAME AND TITLE
S2 == MENU SELECTION ITEMS
CHANGE= S2
FORMATTED MENU PROCDEFS -Screen 2-MENU SELECTION ITEMS Mon Dec 05 11:32:43
1994
MENU.NAME ==> MENU2
No. DESC ACTION EXP
1 @(5,3)SUN.MEMBER INFORMATION REVISE SM USE THIS OPTION TO E
2 @(5,5)LIST SENIOR MEMBERS LIST.SR USE THIS OPTION TO L
3 @(5,7)LIST JUNIOR MEMBERS LIST.JR USE THIS OPTION TO L
4
Change which line item= <Return>
FORMATTED MENU PROCDEFS -Screen 1-MENU NAME AND TITLE Mon Dec 05 11:32:57
1994
1 MENU.NAME MENU2
2 TITLE SUN.MEMBER MENU
3 QUERY ENTER SELECTION NUMBER
4 EXITS EXIT,QUIT,Q
5 STOPS STOP,S,HALT
S1 == MENU NAME AND TITLE
S2 == MENU SELECTION ITEMS
CHANGE= <Return>
FORMATTED MENU PROCDEFS REVISE.1 Mon Dec 05 11:33:03 1994
MENU.NAME= <Enter>
```

Changing the default prompt, exit, and stop options

Field 5 of a menu definition record contains the prompt that displays at the bottom of the menu. If field 5 is empty, the following prompt is displayed by default:

```
Which would you like? (1-n)=
```

Field 6 of a menu definition contains a comma-separated list of characters and words that can be used to exit from the menu and return to the level from which it was called. If you exit from a menu that was called by another menu, you return to the calling menu (one level up). If you exit from a menu called by a command entered at the system prompt, you return to the system prompt. If field 6 is empty, the Enter key is the default exit key.

Field 7 contains a comma-separated list of characters and words that can be used to stop the menu processor and return directly to the calling process. The stop function does not exit to any of the previous menus—it lets you exit from the entire menu system. If field 7 is empty, no stop function is available to the user of the menu.

To change the prompt, the exit function, or the stop function, use option 2 from the **Menu Maintenance** menu.

> **Note:** If you change the default exit, only the new exits that you set can be used with that menu. For example, if you change the exit default to either field, `EXIT` or `QUIT`, a simple Enter no longer performs the exit function. A trailing comma can be used to indicate a Enter as a valid exit.

Nesting menus

You can call another menu from within a menu. This is called nesting menus. Using the **Menu Maintenance** menu, enter the name of the nested menu called in the ACTION field.

When you nest menus, be sure you use consistent commands for leaving the menus. For the default value, EXITS, Enter lets you exit the current menu and returns to the menu that called it. STOPS can be set up so that you exit from the menu and return to the calling process without returning to any of the previous menus. If you specify new exits or stops, use a trailing comma for the Enter. If you have three nested menus and are at the third level, you must press Enter twice to return to the first menu.

Prompting within menus

You can use the inline prompting feature to create menus that display prompts. To do this, use a sentence or a paragraph that contains inline prompting as one of the actions in the menu. UniVerse BASIC programs can pass values for inline prompts to paragraphs with a DATA statement. Prompting from the menu is useful if you do not want to limit the option on the menu to a specific value. For example, instead of this sentence:

```
LIST PAYABLES VENDOR WITH AMT.PD.DATE EQ 10/31/94
```

you could use the following sentence:

```
LIST PAYABLES VENDOR WITH AMT.PD.DATE EQ

<<ENTER LAST PAYMENT DATE>>
```

You can include inline prompts in comment lines also. For more information about inline prompts, see the *UniVerse User Reference*.

You can include a menu selector in the VOC file login record that executes the menu when a users logs on to UniVerse. The menu invoked by the login record might be a master menu listing all the menus in your application. Such a menu is useful for users who are not familiar with the application.

Displaying menus in Motif format

You can use the MOTIF command to display a UniVerse menu in Motif format. This lets you use menus without having to worry about file formats. Since UniVerse prompts you to add the information for System Administration tasks, you do not have to specify parameters in a certain order.

Use the following syntax:

```
MOTIF menu.name
```

menu.name is the record ID of the VOC menu entry that invokes the menu. The options listed on the menu that you invoke must call submenus.

When you display a menu in Motif format, the title of the menu is at the top of the screen. Under the title, options of the main menu appear in a menu bar. The first menu option is highlighted. Each option stands for a pull-down menu that scrolls down from the main menu bar when you select an option.

Menus list actions that can be performed and options that display submenus (cascading menus). Options followed by an arrow (=>) display a submenu.

The options on the submenus are names or descriptions of UniVerse sentences that are stored as part of the menu. This eliminates the need to remember sentence names.

To display a submenu, highlight the option, then select it. To start an action, simply select it. If a data entry screen appears, instructions in the lower part of the screen prompt you to enter the appropriate data. System messages also appear at the lower part of the screen.

Responding at any level of a Motif menu system

Certain responses work at all levels of a Motif menu system. At any menu bar, use the Right Arrow key and Left Arrow key to move the highlight to the option you want.

Choosing an option

To choose the highlighted option, press Enter or Spacebar. To move to an option and choose it with one keystroke, press the capitalized letter (mnemonic) of the option you want. The mnemonic is not always the first letter of an option. If your keyboard does not have arrow keys, you can use the mnemonic letters to move around the menu system and make selections.

Moving around the menus

To move the cursor up and down in the submenus, use the Up and Down arrow. Note that the up arrow does not work on menu bars, and the down arrow works only if there is a submenu. To move from a submenu to the one immediately above it, press the Left arrow. To return directly to the main menu level from any submenu, press F10.

To exit a Motif menu or submenu and return to the UniVerse prompt, press Esc.

Examples

This example shows the `LIST.FILES` menu in Motif format:

```
>MOTIF LIST.FILES
```



The next example shows the `LIST.FILES` menu in standard UniVerse format:

```
>LIST.FILES
Available Files
```

```
1. ORDERS
2. CUSTOMERS
3. BOOKS

Which would you like? ( 1 - 3 ) ?
```