

SD

...

The

Open Source

Multi-value

String

Database

Preface

SD, the Multivalue String Database

SD is a multivalue database in the Prime Information tradition. It contains open source code from the Open Source databases openQM (revision 2.6.6) and ScarletDME and open source code developed by the SD developers after the fork from ScarletDME. While it shares many of the same features, it was forked to explore some new ideas as to what a modern multi-value database should contain.

OpenQM is currently copyright to Rocket Software. At the time of the Open Source release openQM was copyright to Ladybridge Systems. This copyright covers all aspects of *OpenQM* including source code, executable code, and documentation.

Acknowledgments

The majority of the content of Section 1 “A Multi Value Primer”: originated with the book “*Getting Started in OpenQM – Part 1 and Part 2.*” The Getting Started in OpenQM series contains the following copyright notice:

“This book is copyright to Rush Flat Consulting (2008-2013).

However, this book may be freely copied and distributed provided that the copyright to Rush Flat Consulting remains in place.

Similarly, portions of this book may be freely quoted provided that Rush Flat Consulting is acknowledged as the source of the quoted material.”

A special thanks goes out to Brian Speirs, the author of the Getting Started in OpenQM series for allowing us to use this content in this manual.

Warning and Disclaimer

Every effort has been made to make this book as complete and accurate as possible, but no warrant or fitness is implied. The information provided is on an “as-is” basis. The author shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this manual.

Table of Contents

SECTION ONE A Multi Value Primer.....	7
What is a multi-value database?.....	7
Non-conformity to relational rules.....	7
Loose data typing.....	9
Data storage.....	9
Hashed files.....	10
Built in programming language.....	10
Built in reporting language.....	11
Summary.....	11
Multi-Value Terminology.....	12
Accounts.....	12
Users.....	12
Database files.....	12
Program files (or directory files).....	13
The VOC file.....	14
Records (or items).....	14
Fields (or attributes), values, and sub-values.....	14
A Note On Capitalization.....	14
Command Variations.....	15
Conventions In Manual.....	15
Multi-value File Concepts.....	16
The data portion.....	16
The dictionary portion.....	16
Creating and Deleting Files.....	18
Standard files.....	18
Directory files.....	18
Multi-part files.....	18
Single level files.....	19
Q-Pointers.....	20
Listing the files.....	21
Creating an Example Database.....	22
Create a file.....	22
Prepare the data.....	23
Create some dictionary items.....	23
Import the data.....	25
Add another file to the database.....	26
Add a third file to the database.....	28
Editors.....	29
The command stack and command editing.....	29
The dot commands.....	30
Editing keys.....	31
Preserving the command stack between sessions.....	32
SD Database Files.....	33

Background.....	33
Hashed Files.....	33
Traditional hashed files.....	33
Dynamic hashed files.....	36
Analyzing a file.....	37
Setting or changing parameters.....	39
Example file configuration.....	40
Directory Files.....	43
Data Storage.....	44
Variable length fields.....	44
String representation.....	45
Internal Data Storage.....	46
Binary Data.....	48
File VOC Entries.....	49
Basic concepts.....	49
Different types of VOC entries.....	51
Dynamic and directory files.....	51
Multi-files.....	51
Q-Pointers.....	52
Manual creation of F-type VOC entries.....	53
Alternate Key Indices.....	54
Summary.....	57
SDQuery.....	57
Anatomy of a <i>SDQuery</i> Statement.....	58
General syntax.....	58
Selection clause.....	58
Creating a dictionary item for use in selecting data.....	59
Multiple selection criteria.....	63
Comparison against a database value.....	64
Direct identification of items.....	64
Sort clause.....	65
Display clause.....	66
Modifiers.....	67
Grouping records.....	67
Generating summary information.....	69
MIN, MAX, AVG.....	70
Suppressing detail lines.....	71
Formatting column headings.....	71
Totaling data.....	75
Page breaks.....	77
Grand totals.....	78
Scaling data.....	79
Page headings and footings.....	80
Printing and Report Styles.....	82
Printing.....	82
The LPTR keyword.....	83
Print units.....	83

Initialising print units.....	85
Defining your own print units.....	85
Spooling print files.....	86
Deleting print files.....	86
Miscellaneous Aspects of SDQuery.....	87
Default display and phrases.....	87
Saving SDQuery statements for later use.....	89
Introduction to SDBasic.....	91
General Considerations.....	91
What is SDBasic.....	91
What is covered here?.....	91
Coding styles.....	91
Where is the GUI?.....	93
General Programming Issues.....	93
Creating, Compiling, and Running Programs.....	93
Statements, variables, tokens, constants, and operators.....	95
Statements.....	95
Variables.....	96
Tokens.....	97
Constants.....	97
Multi-value Variables.....	98
Operators.....	99
Substring extraction.....	99
Pattern matching.....	100
Alternative relational operators.....	100
Assignment.....	101
Assignment shortcuts.....	101
Substring assignment.....	101
Null values.....	102
A simple program.....	102
Program.....	103
Analysis.....	104
Detail points.....	105
Some useful functions and statements.....	109
CRT and DISPLAY.....	109
PRINT.....	110
INPUT.....	110
DCOUNT.....	111
FIELD.....	111
ICONV / OCONV.....	112
LOCATE.....	113
FIND.....	118
Program control structures.....	119
IF-THEN-ELSE.....	119
CASE.....	120
Loops.....	121
Conditional loops.....	121

For-Next loops.....	122
EXIT and CONTINUE.....	124
Subroutines.....	124
Program structure.....	125
Internal subroutines.....	126
External subroutines.....	126
Local subroutines.....	129
User Defined Functions.....	130
Files.....	131
Opening files.....	132
Error handling.....	133
Selecting data in files.....	135
Internal select.....	135
External select.....	136
Which selection should I use?.....	137
Reading from files.....	137
Getting the ID from the select list for the READ.....	138
Writing to files.....	139
Closing files.....	139
Other methods of file handling.....	139
Multi-user issue.....	140
When should locking be used.....	141
File locks.....	141
Record locks.....	142
SECTION TWO Getting Started With SD.....	145
Installation - Debian 12 with Gnome or Ubuntu 24.04.....	145
Installation on server without a GUI.....	148
Installation on WSL.....	149
Configuration.....	150
SD Connection Methods.....	152
SD User and Group Accounts.....	156
TCL – The Command Line.....	159
Encryption in SD.....	168
Embedded Python in SD.....	172
SECTION THREE Developer Notes.....	175
Changing Revision.....	175

SECTION ONE A Multi Value Primer

What is a multi-value database?

Firstly, multi-value databases have been designed from the ground up as multi-user databases.

Multi-value databases have a number of characteristics that make them different from relational databases such as *mySQL*, *SQL Server*, or *Oracle*.

These are:

- their data model does not (have to) conform to relational rules
- data is loosely typed
- data is stored in literal format, in variable length records
- data is stored in hashed files
- they come with their own in-built programming language
- they come with their own in-built reporting language allowing fully formatted reports to be generated from the data

Modern multi-value databases also provide interfaces to external programming languages, socket connections, and the ability to interact with the host operating system.

What do these things actually mean?

Non-conformity to relational rules

Take the example of a typical invoice. In a relational database, invoice data is stored in two tables. The first table represents the invoice header and contains the invoice number, date, and customer reference (among other things). The second table contains the line details of the invoice. This second table is linked back to the invoice header by way of the invoice number. This structure is shown below:

Invoice number	Date	Customer number
12345	24 Apr 2007	9854
12346	24 Apr 2007	6234
12347	25 Apr 2007	4921

Invoice detail number	Invoice number	Product ID	Quantity	Price
671245	12345	9854	2	15.00
671246	12346	6234	1	32.50
671247	12346	4921	1	23.90
671248	12347	5651	3	12.50
671249	12347	5694	2	3.50
671250	12347	6234	5	32.50

These tables show the data for three invoices. The first invoice has one line item, the second has two line items, and the third has three line items. The data in the second table (the line items) can be related back to the correct customer through the invoice number.

In a multi-value database, all this data could be contained in just one table (referred to as a file). The structure of the multi-value invoice file is shown below:

Invoice number	Date	Customer number	Product ID	Quantity	Price
12345	24 Apr 2007	9854	9854	2	15.00
12346	24 Apr 2007	6234	6234	1	32.50
			4921	1	23.90
12347	25 Apr 2007	4921	5651	3	12.50
			5694	2	3.50
			6234	5	32.50

If you examine these two structures, you will see that the single multi-value structure contains all the same information as the two structures in the relational database. The difference is that each of the Product ID, Quantity, and Price fields have multiple entries in the field.

Note that it is much quicker to read an invoice from a multi-value database than from a relational database. Using invoice 12347 as an example, a multi-value database can read this invoice with a single disk read¹. In comparison, a relational database would use 4 disk reads for the data, plus a few more for reading indices.

¹ This assumes that the application knows the ID (primary key) of the invoice. If this is known, the database will calculate the location of the item, and read it in a single read.

Loose data typing

In many databases, fields are defined as being of a specific data type, and the database will not allow data of any other type to be stored in that field. Multi-value databases do not follow this pattern.

Firstly, database fields do not actually need to be formally defined. Of course, well structured databases do have field definitions, but even then, the definitions are descriptive rather than prescriptive.

Secondly, even if the field definition says the data is of a certain type, the database itself places no restrictions on the type of data actually entered into the field. Therefore, string data may be entered into a numeric data field and vice-versa with no objections² from the database³.

Thirdly, in the programming language, the typing of variables is not required, and may change within the program. For example:

```
Temp = 0
...
...
Temp = 'Q'
```

These characteristics mean that multi-value databases are flexible, and make it easy to accomplish certain tasks. The flip side of this coin is that it is easy to end up with a database structure that is (a) not defined, (b) only partly defined, or (c) incorrectly defined. Likewise, it is possible to end up with unexpected data types in the database fields.⁴

Data storage

As noted above, many databases get you to define the field types. One of the reasons they do this is so that numbers can be stored as a numeric data type. What this means is that the integer 123 can be stored as a single byte representation. On the other hand, those databases will typically reserve either 4 or 8 bytes for an integer value, even though only one byte is being used.

Multi-value databases store data as literal strings. Therefore, 123 is stored as the string “123”, and the field length is 3 characters. If the number changed to “12345” then the field will be expanded to 5 characters.

These variable length fields are achieved by using special characters to delimit the fields in the record. The database counts along the delimiters to find the requested field, value, or sub-value.

Consider an address database. The name and address data fields will typically be 30 characters long in a traditional database. With one name field and four address lines, the record will consume 150 bytes, regardless of much data is actually stored in that space. A multi-value database will use as many bytes

2 No objections from a storage perspective. However, you may have difficulties processing the data using the programming language if you mix data types.

3 Data type integrity could be enforced by using triggers on the data file to test the data before it was written to disk.

4 The phrase “Give them enough rope” is often used in discussions regarding multi-value databases. The flexibility and lack of enforcement of rules make it easy to create poorly structured databases, maintained by poorly structured code. Basically, the integrity of the system is in the hands of the developer(s).

as are entered, plus the delimiter characters (5 in this case). If a line is longer than 30 characters, then the multi-value database is able to store the extra characters (where a traditional database cannot) although you will then have an issue of how to print them if you are restricted to 30 characters on an address label. If a line isn't used, then the multi-value database will only store a delimiter character.

When storing strings like an address, a multi-value database may only require a third the disk space that a relational database would use because it doesn't need to reserve space for fields, it simply uses space when it is required. When storing numbers, it is more evenly balanced, with the multi-value database using more space for large numbers and less for small numbers.

Hashed files

A hashed file consists of a series of groups or buckets. Records are assigned to a group using a pseudo-random method based on the record ID. This is how a multi-value database can find a record quickly if the ID is known. The process is:

- The ID is hashed to form a large number.
- The large number is divided by the modulo (number of groups) of the file. The remainder from the division is the group number.
- The entire group is read from disk, and the record is found by searching through the group.

The combination of hashed files and storage of records as variable length strings make for a highly efficient storage and retrieval system. See chapter SD -Files for more on the theory and practice of hashed files in greater depth.

Built in programming language

Many databases (such as MySQL) do not provide a programming language to access or manipulate the data in the database. Rather, they provide an 'Application Programmer Interface' (API) which allows an external programming language (such as Perl, Python, PHP, or Visual Basic) to access the database. This means that programmers can use a language with which they are familiar – if there is an API for that language, and that the database provider does not have to put resources into developing and maintaining a programming language.

Multi-value databases have a different approach. They tend to provide an entire database environment including a programming language and reporting facilities. This is particularly useful for exploiting the multi-dimensional nature of the database.

The actual language provided is a dialect of BASIC. This has been extended to be used with multi-value data in a multi-user environment. It is simple to learn, but contains powerful data and string handling capabilities.

Modern multi-value databases often also incorporate one or more API's for use with external languages. Typical API's are for C, Visual Basic, and/or ODBC.

Built in reporting language

Multi-value databases incorporate a combined query and reporting language that allows you to:

- report on data contained in one or more files
- select records to be reported on (multiple selections)
- sort the data by multiple sort criteria
- break the data into groups
- create data columns derived from other data
- calculate totals, averages, and percentages
- format the data to display in a specified format
- format the report with headings, footings and page breaks
- and more

This is all achieved through a sentence based query language. For example:

```
SORT INVOICES WITH DATE GE "01-04-2013" AND LE "31-04-2013" BY CUSTNO BREAK-ON
CUSTNAME "'UV'" ENUMERATE INVNO INVDATE TOTAL AMOUNT HEADING "'DGC'Invoices
for April 2013'G'Page 'PL'" FOOTING "Monthly invoice report" ID.SUP
NO.GRAND.TOTAL LPTR
```

This would select all invoices for April 2013, sort them into Customer No order, and produce a report showing the customer name, and the invoice number, date and amount of each invoice. After all invoices for each customer have been displayed, totaled and enumerated fields will be underlined and the count of invoices, and the total amount of the invoices for that customer will be displayed. Pages have a defined heading and footing, and the report will be sent to the printer.

Summary

Overall, multi-value databases are flexible and easy-to-use. The combination of an easy to use programming language and reporting/query language that allows flexible reporting is a powerful combination.

Interfaces to external languages allow multi-value databases to be incorporated seamlessly into a Windows (or linux) environment, although this means that you forgo most of the inbuilt reporting capabilities.

Multi-Value Terminology

Multi-value databases have their own terminology. This section provides a quick coverage of that terminology and places it in a relational database and general computing framework.

Accounts

Information in multi-value databases is organized into accounts. An account is loosely analogous to a database in relational terms.

In *SD*, an account is implemented as a folder or directory, with the database files implemented as folders or sub-directories within that account.

When logging on to *SD*, a user is automatically logged into their home account (unless the `-a<account name>` option is present). They then have direct access to all the database files in that account. Data from files in other accounts can be accessed using a file pointer. This makes the data appear as if it is local to the current account. Users can log between accounts at will (security allowing).

Overall, an account is simply a way to group related database files.

Users

A user must be registered to use *SD*. *SD* uses the operating system authentication to validate a user name, although it is possible to add further restrictions on users once they have entered the *SD* environment.

It is important to note the distinction between users and accounts. Accounts may be used by many users (simultaneously), and individual users may use multiple accounts. Individual users may have multiple concurrent sessions, in one or more accounts. On the other hand, some users will only use a single account, and may be restricted to a single account even when multiple accounts are available.

Database files

A database file is analogous to a table in a relational database. Whereas a relational database is made up of multiple tables, a multi-value database is made up of multiple files.

A multi-value database file normally consists of two parts – a dictionary, and a data portion – although each part can exist independently of the other, and a dictionary may be associated with multiple data portions.

A file dictionary exists to provide definitions of the data in the data file(s) *for reporting purposes*. This qualification is important because:

- the dictionary does not define or restrict the data in the manner of a relational database. The dictionary is purely descriptive
- the description is not enforced (by the database) and does not have to be correct!
- a data element can be described in multiple ways. For example, a numeric field may have three definitions to show its value in units, thousands, and millions
- the primary use of the definitions contained in the dictionary is for reporting purposes using the *SDQuery* reporting language.

In *SD*, database files are implemented as operating system folders or subdirectories. Dictionary and data portions of the file each have their own folder, the data portion will take its folder name from the filename you specify in *SD*, while the dictionary folder will have '.DIC' added to the *SD* filename.

For example, assume that we are in an account named TEST which has been created in folder /home/sd/group_accounts/TEST. If we create a normal file named TESTFILE, *SD* creates two sub-folders beneath the TEST folder named TESTFILE and TESTFILE.DIC.

In normal files, each of the dictionary and data folders contains two files named '%0' and '%1'. These are hashed files, where *SD* maintains the filing structure and indices. While these files are visible at the operating system level (e.g. through a file manager), the contents of these files should not be edited using any operating system utilities.

In the case where a dictionary is associated with multiple data portions, the 'data' folder holds a sub-folder for each data portion.

Program files (or directory files)

Traditionally, multi-value databases only used hashed files as described above. However, more modern implementations such as *SD* recognize that hashed files are inefficient storage mechanisms for items such as programs, and accordingly have implemented the use of directory files for these types of items.

A directory file still consists of dictionary and data portions. However, the data portion is simply an operating system folder. The dictionary portion continues as a hashed file as described above.

Items contained in a directory file can be accessed and edited directed from the operating system environment as well as from within the *SD* environment.

While directory files can be used to hold "normal" structured database data, this is not recommended. Data retrieval from directory files is much less efficient than from hashed files.

Typical uses of directory files are to store programs, images, or PDF files.

The VOC file

Multi-value databases provide a command driven environment. Accordingly, the database must be able to understand the commands issued by a user. It does this by storing command definitions in a special file known as the voc or vocabulary file.

Each account has its own voc file. This means that System Administrators can restrict actions within certain accounts by removing selected keyword definitions, or that commands can be added to the voc that are relevant to that account.

Records (or items)

SD uses the terminology of records, fields, values, and sub-values.

A multi-value record is loosely analogous to a record in a relational database. The key difference is that a multi-value record can be equivalent to a *group of records* in a relational environment. If you refer back to Chapter **What is a multi-value database**, section **Non-conformity to relational rules**, you can see that invoice number 12347 holds the data equivalent of several relational database records. This is why an item can be thought of as a group of records.

Records also refer to programs. For normal programming languages, a program is an individual file held within a folder. In the multi-value world, a program is a record within a file (a programs file). In *SD*, you can actually have both views of programs, because if the programs are held in a directory file, then *SD* will see them as records in a file, but the file manager will see them as files within a folder.

Fields (or attributes), values, and sub-values

Multi-value fields (attributes) are loosely equivalent to fields in a relational database. However, attributes can be divided up into several values, which in turn can be divided into several sub-values. Relational databases have no equivalent of values and sub-values, and need to use multiple tables to store the equivalent data structures.

Section **Non-conformity to relational rules** has already shown how the 'Product ID' field on the invoice can hold several data values. This is the core principle of the multi-value database.

Examples of the use of sub-values are less intuitive. Consider sub-values as a way of storing a bit more related data.

A Note On Capitalization

Multi-value databases originated at a time when data entry was largely restricted to upper case characters. These early databases only recognized commands entered in upper case. Likewise, all programming was required to be in upper case.

As time went by and usage of mixed case became prevalent, different databases adapted in different ways. *SD* attempts to be reasonably case insensitive:

- The programming language is not case sensitive.
- When a command is entered at the keyboard, or *SD* searches for dictionary items, it firstly attempts to find the command or dictionary item in the case as typed. If the command or dictionary item is not found, then the word is converted to upper case and the search repeated.

To illustrate the significance of the second point, consider that you have created a dictionary item named 'myDictItem'. If you subsequently reference that dictionary item in any command or query, then you will need to type it exactly as you have named it. However, if you name the dictionary item in upper case (i.e. 'MYDICTITEM'), then *SD* will find the dictionary item *whatever case you use in your query command*.

For this reason, it is recommended that you name all voc items, dictionary items, and programs in UPPER CASE.

Command Variations

SD accepts a number of command variations. These variations allow for:

- North American word spelling as well as British spelling (e.g. CATALOG is accepted as being the same command as CATALOGUE)
- compatibility with other multi-value databases (e.g. ID-SUPP is accepted as a synonym for ID.SUP).

Conventions In Manual

SD is a command-driven environment.

Where the syntax of commands is shown, curly brackets⁵ denote optional components, while a pipe symbol denotes that only one of the separated options should be used:

```
SEARCH {DICT} filename {ALL.MATCH | NO.MATCH} {NO.CASE}
```

⁵ In many computer manuals, optional components are often shown in square brackets. However, as square brackets are used within the *SD* query language and programming language, curly brackets are used both here and in the documentation to avoid confusion.

Multi-value File Concepts

It was noted earlier that multi-value files generally have two parts – a dictionary and a data portion. What is the significance of this structure?

The data portion

The data portion contains (surprisingly enough) – data.

Each record in the database is identified by a unique identifier. In multi-value terms, this is called the `ID` or item-id, and is equivalent to the primary key in relational databases. The real power of the `ID` comes from its use as a data locator within the database.

As long as the database is given the `ID` of a record, then the database can (usually) read the record in a single disk read – even when the database has not been indexed. This characteristic has long been used to provide high performance in multi-value systems – by making the `ID` of a file ‘meaningful’ (such as a customer number or part number), then the database can find the associated data very quickly.

An example of non-meaningful data as an `ID` is a sequential number (auto increment field). This is simply an `ID` that is assigned to the record, and has no relationship to the data contained in the record.

The choice of the `ID` for each record thus becomes an important decision. There are arguments both for and against using meaningful data as the `ID`. In general, meaningful data should only be used if:

- it is NEVER going to change
- it will always be unique.

Therefore, a customer number is good, but a customer surname is not.

These rules may seem simple, but in real life you will often find data that does not conform to expectations. Duplicate numbers will exist in data where you expect the number to be unique, and “permanent” numbers will change over time. Part numbers are notorious for this type of duplication and change.

In a complex system, changing the primary key is a non-trivial exercise. Therefore, take care in the initial design stage of the database. If in doubt, use a sequential number and index the database on the key fields.

The dictionary portion

The dictionary portion contains data descriptions. *SDQuery* (covered later in this manual) uses these descriptions to extract and display the data.

The following points may be of use in understanding dictionaries:

- Dictionaries are not a schema, although they should “describe” the data
- Dictionaries are not compulsory
- The descriptions they contain may not be accurate
- You can have multiple descriptions for each field
- Dictionary items can contain complex calculations as well as general formatting instructions
- Dictionary items can look up data in other files on the system
- It is up to you as the user/administrator to define what goes in the dictionary.

Notwithstanding the above, the dictionary SHOULD represent the data in the data file. Maintenance and understanding of the system is enhanced immeasurably if the dictionary is complete and up to date. Therefore, it is highly recommended that you use and maintain dictionaries to document the database.

Dictionaries may contain the following types of items:

- D Direct data items. These items describe the data in the file.
- I Indirect data items. These items calculate a new value from the data in the file.
- L Link items. These items join the current file to another file.
- PH Phrases.
- C Calculated data values. These items contain an embedded *SDBasic* program, and are used to generate calculated values.
- A A PICK style attribute defining item.
- S A PICK style synonym item.
- X Other miscellaneous data.

The dictionary type (one of the above values) is declared in the first field of the dictionary item. This manual will cover the first four of these dictionary types.

In the next part of this section, we will create a few dictionary items. We will use those dictionary items to import and work with some data. But to fully understand dictionaries, you will need to read the sections on *SDQuery* and *SD Dictionaries* – and then look the at *SD* manuals to provide greater depth than will be covered here.

Creating and Deleting Files

Standard files

Files store the data you want to work with. The command to create a file in *SD* is:

```
CREATE.FILE filename
```

The `CREATE.FILE` command as shown creates a two-part file, a dictionary part, and a data part. In some situations, you may only want to create a dictionary or a data portion. The commands to do this are covered in the section on multi-part files below.

The command to delete a file is:

```
DELETE.FILE filename
```

Both of these commands have variants that allow individual parts of a file (a dictionary or data portion) to be created or deleted independently of the other parts. These variants are normally used to maintain multi-part files, a dictionary file that is associated with multiple data files. These are covered later in this section.

Directory files

Creating a directory file is accomplished using the following command:

```
CREATE.FILE filename DIRECTORY
```

As you can see, this is just a small variation on the command to create a standard (hashed) file.

Note that only the data part is created as a directory file, the dictionary is still created as a dynamic hashed (standard) file. In a directory file, all of the file items (records) are created as individual operating system level files rather than being contained within a single dynamic file. Directory files are usually used to store basic programs.

Directory files can be deleted using the standard `DELETE.FILE` command.

Multi-part files

Multi-part files need to be created when you want multiple data files to use a single dictionary. Typical uses for this type of structure are for archiving data, or keeping data from individual years separate.

When creating a multi-part file, each individual part of the file is created with a separate command:

```
CREATE.FILE DICT dictname  
CREATE.FILE DATA dictname,dataname
```

The first command creates just a dictionary portion, and the command structure is quite obvious. However, the command to create a data portion needs a little more explanation.

In order to associate the data file with a specific dictionary, *SD* must know which dictionary name to use. Therefore, the dictionary name is entered as part of the command. Note the comma that separates the dictionary name from the data file name.

SD can change ordinary files to multi-part files if you add a second data part to it using the data command above.

Deleting parts from multi-part files uses a similar format command to that used for creating the parts:

```
DELETE.FILE DICT dictname
DELETE.FILE DATA dictname,dataname
```

Note that *SD* allows you to delete the dictionary of a multi-part file without deleting the data portions. Under normal circumstances, this would be an unusual thing to do. But perhaps the dictionary didn't actually contain anything so served no purpose on the system.

Consider the following examples of creating and deleting files:

```
CREATE.FILE TEST
Created DICT part as TEST.DIC
Created DATA part as TEST
Added default '@ID' record to dictionary

CREATE.FILE DATA TEST,TEST2
TEST already exists but not as a multifile
Convert to multifile named "TEST,TEST" (Y/N)? Y
Created DATA part as TEST\TEST2

DELETE.FILE DICT TEST
DICT portion 'TEST.DIC' deleted

DELETE.FILE DATA TEST
Delete all data components of multifile? Y
OK to delete DATA portion 'TEST\TEST'? Y
DATA portion 'TEST\TEST' deleted
OK to delete DATA portion 'TEST\TEST2'? Y
DATA portion 'TEST\TEST2' deleted
Multifile directory 'TEST' deleted
VOC entry 'TEST' deleted
```

In this sequence, an initial file (TEST) is created. This file is then converted to a multi-part file with the addition of a second data portion (TEST2). The dictionary is then deleted, followed by the two data portions.

Single level files

Sometimes, you may want only a data file without any accompanying dictionary. Such a file may be used for control purposes, and you do not want to do any reporting on the file using *SDQue/ry*.

The command format is similar to that used for multi-part files:

```
CREATE.FILE DATA dataname {DIRECTORY}
```

To delete such a file, you can use either the standard or the multi-part DELETE.FILE format. If you use the standard format, *SD* reports that the dictionary part of the file does not exist:

```
CREATE.FILE DATA TEST
Created DATA part as TEST
```

```
DELETE.FILE TEST
DATA portion 'TEST' deleted
DICT part of file does not exist
VOC entry 'TEST' deleted
```

Q-Pointers

A Q-type record has three general purposes.

The primary one is to reference a file in another account. By placing the Q-type record in the voc of the current account, you can then reference the remote file as if it were a local file.

The second reason to use a Q-type is to create a synonym name or an alias for the target file. For example, if you have a main file INVOICES, which you archive by financial year to an annual data file associated with the INVOICES dictionary, then the full file name for the 2011-12 invoices could be: INVOICES,201112 This is inconvenient to type, so you may want to create an alias for this file – e.g. INV11.12 This is much easier to use in a *SDQuery* statement.

The third reason to use a Q-type is to reference a file on a remote *SD* server.

The best way to see what a Q-pointer does is to see one work.

One of the files in the SDSYS account is named BP (for Basic Programs). How do we access the contents of that file from our current account ? We create a Q-pointer from our current account to that file. Let's call the Q-pointer BP.SDSYS.

We create the Q-pointer in the voc of our current account, using one of the *SD* editors:

<pre>ED VOC BP.SDSYS VOC BP.SDSYS New record ----: I 0001= Q 0002= SDSYS 0003= BP 0004= Bottom at line 3 ----: FI 'BP.SDSYS' filed in VOC</pre>	<pre>Invoke the editor SD responds with this and this Type I to go into insert mode Type Q to indicate a Q-pointer This is the account name This is the file name Press enter to exit insert mode SD responds with this Type FI to file the item SD responds with this</pre>
---	--

Now, test the Q-pointer:

```
SORT BP.SDSYS
Page 1
BP.SDSYS...
BIGSTR_TEST
MSGTEST
PCL
PCL.GRID
PCODE_LIST
PY_TEST
SDTEST_V8
SD_ENCRYPT
SD_ENCRYPT_B
64
SD_ENCRYPT_E
XT
SD_EXT
TEST.THEN.EL
SE
TESTSZ
U0032
U50BB
VFS.CLS
sdTests
```

17 record(s) listed

We can see here that a Q-pointer can be used to access a remote file. The same concept can be used to access a local file using a different name.

While a Q-pointer is not a real file, it accesses a real file. You can create, modify, and delete items in the real file by using the appropriate commands on the Q-pointer.

Many people have got into problems because they deleted items in a Q-pointer file, thinking that they were only deleting copies of the items. In fact, they deleted the real items in the remote file – so be careful with Q-pointers.

To remove the Q-pointer, you simply delete the voc entry:

```
DELETE VOC BP.SDSYS
```

Listing the files

There are four commands available to list the files that are available within an *SD* account:

- LISTF List all F-type records in the voc
- LISTFL Show only local files (in the account)
- LISTFR Show only remote files (referenced in the VOC but not in the account)
- LISTQ List all Q-type records in the voc

Files accessible by *SD* generally need either an F-type or a Q-type record in the voc. What does that mean?

When a file is created, the `CREATE.FILE` command will automatically create an F-type record in the voc. This lets other *SD* commands “find” the file when you reference it in *SDQuery* statements or *SDBasic* programs.

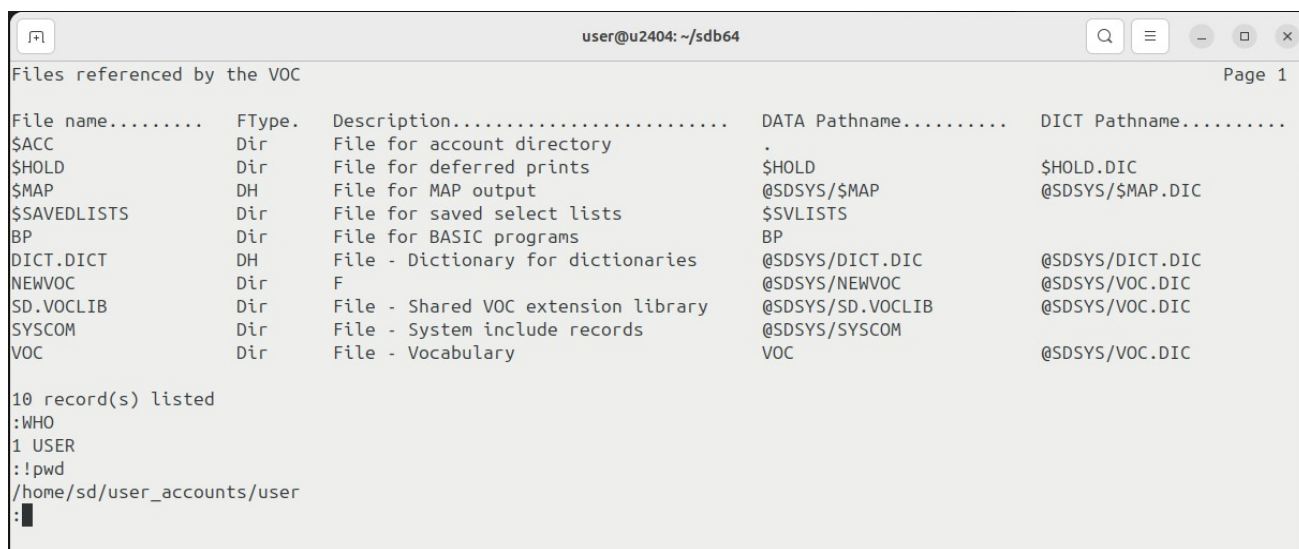
In essence, an F-type refers to a file within the account, while a Q-type is a pointer to a file which is (usually) in another account.

There is another category of files called 'remote' files. These files aren't local to this account, but may not be in any other account either. For example, we could set up an F-type item that points to a /tmp folder. Because the location /tmp is not within the local account, this is considered a remote file.

SD comes with a few inbuilt remote files. You can list these with the LISTFR command. These are largely system files.

By default, remote files are not used very much. However, if you, as a developer, wish to separate the data components of an application or an account from the software components, you may find yourself using remote files extensively.

Typing in any of the file listing commands will provide an output similar to those shown below:



```
user@u2404: ~/sdb64
Files referenced by the VOC
Page 1
File name..... FType.  Description.....  DATA Pathname.....  DICT Pathname.....
$ACC             Dir     File for account directory      .
$HOLD            Dir     File for deferred prints        $HOLD
$MAP             DH      File for MAP output             @SDSYS/$MAP
$SAVEDLISTS     Dir     File for saved select lists     $$VLISTS
BP              Dir     File for BASIC programs         BP
DICT.DICT       DH      File - Dictionary for dictionaries @SDSYS/DICT.DIC
NEWVOC          Dir     F                                @SDSYS/NEWVOC
SD.VOCLIB       Dir     File - Shared VOC extension library @SDSYS/SD.VOCLIB
SYSCOM          Dir     File - System include records    @SDSYS/SYSCOM
VOC             Dir     File - Vocabulary               VOC
$HOLD.DIC       $HOLD
@SDSYS/$MAP.DIC @SDSYS/$MAP
$SAVEDLISTS     $$VLISTS
BP              BP
@SDSYS/DICT.DIC @SDSYS/DICT.DIC
@SDSYS/NEWVOC   @SDSYS/NEWVOC
@SDSYS/SD.VOCLIB @SDSYS/SD.VOCLIB
@SDSYS/SYSCOM   @SDSYS/SYSCOM
@SDSYS/VOC.DIC @SDSYS/VOC.DIC

10 record(s) listed
:WHO
1 USER
:!pwd
/home/sd/user_accounts/user
:
```

Creating an Example Database

Create a file

Before we create a file, we need to ask:

- what are we going to store in the file
- what are we going to call the file.

There are lots of things to consider about the structure too, but that can wait until after we've created the file.

Our first file will contain a series of exchange rates. We can download these in Excel spreadsheet form from: <https://www.rbnz.govt.nz/statistics/series/exchange-and-interest-rates/exchange-rates-and-the-trade-weighted->

index. Once you are on that page, download the file of daily exchange rates marked: 'Exchange Rates and TWI B1 Daily (2018 to current)'. The file name is: 'hb1-daily.xls'.

As for a name, we could give the file a long name like EXCHANGERATES, or EXCHANGE.RATES, but for convenience, we'll use something a little shorter – XRATES. Therefore, our command to create the file is:

```
CREATE FILE XRATES
Created DICT part as XRATES.DIC
Created DATA part as XRATES
Added default '@ID' record to dictionary
```

The computer responds by telling us that it has created the dictionary (DICT) and data (DATA) parts of the file, and has added a record named @ID to the dictionary.

Prepare the data

The data we downloaded isn't in the best layout for importing into *SD*. So, we'll restructure that data so it is ready for importing. Open the file using any spreadsheet program that can read Excel files. The data we want starts in cell A6 .

We want to several things with this data:

- strip out the unnecessary rows (1 - 5)
- save the spreadsheet in csv format (Caution NOT in.CVS UTF-8)

Now save the spreadsheet as a CSV file. Choose 'File | Save as' from the menu, select a file type of csv, and save with the original base file name. This should save the file as 'hb1-daily.csv'.

Create some dictionary items

We want to create a dictionary item for each of the exchange rates in the file. To do this, we are going to use the MODIFY editor within *SD*:

```
MODIFY DICT XRATES
```

The editor will respond: **Id (? for list):**

Type in: **DATE**

A list of fields will now be displayed:

```
Id (? for list): DATE
1: TYPE/DESC=
2: LOC      =
3: CONV     =
4: NAME     =
5: FORMAT   =
6: S/M      =
7: ASSOC    =
```

And a prompt will appear at the bottom of the screen:

TYPE/DESC:

This is asking what type of dictionary item are we going to create. Answer: **D**

The prompt changes to **LOC** This is the field number. Answer: **0⁶**

The remaining prompts and answers are:

CONV	D
NAME	Date
FORMAT	11R
S/M	S
ASSOC	

Just press enter at the ASSOC prompt to leave it blank. The following prompt will now appear:

Action(n/FI/Q/?):

If everything is correct, type **FI** to file the item. Otherwise, enter the line number of the entry you would like to correct. Once the item is correct, enter **FI** to file it.

Now create the following dictionary items:

ID	Type	Loc	Conv	Name	Format	S/M	Assoc
USD	D	1	MR44,	US Dollar	7R	S	
GBP	D	2	MR44,	UK Pound	7R	S	
AUD	D	3	MR44,	Aus Dollar	7R	S	
JPY	D	4	MR22,	Jap Yen	7R	S	
EUR	D	5	MR44,	Euro	7R	S	
CAD	D	6	MR44,	Canada Dollar	7R	S	
KRW	D	7	MR22,	SKorea Won	7R	S	
CNY	D	8	MR44,	Chinese Yuan	7R	S	
MYR	D	9	MR44,	Malay Ringgit	7R	S	
HKD	D	10	MR44,	HK Dollar	7R	S	
IDR	D	11	MR22,	Indonesia Rupiah	9R	S	
THB	D	12	MR44,	Thai Baht	8R	S	
SGD	D	13	MR44,	Singapore Dollar	7R	S	
TWD	D	14	MR44,	Taiwan Dollar	7R	S	

To exit the editor, simply press enter when it prompts you for a new ID. Now to see your completed dictionary items, type:

```

SORT DICT XRATES
Page 1
@ID..... TYPE LOC..... CONV.. NAME..... FORMAT S/M ASSOC...
DATE      D    0      D      Date      11R    s
@ID      D    0      X      XRATES   10L    S
USD       D    1      mr44,   us dollar 7r     s
GBP       D    2      mr44,   uk pound  7r     s
AUD       D    3      mr44,   aus dollar 7r     s
JPY       D    4      mr22,   jap yen   7r     s
EUR       D    5      mr44,   euro      7r     s
CAD       D    6      mr44,   canada dolla 7r     s
          r
KRW       D    7      mr22,   skorea won 7r     s
CNY       D    8      mr44,   chinese yuan 7r     s
MYR       D    9      mr44,   malay ringgi 7r     s
          t
HKD       D   10      mr44,   hk dollar  7r     s
IDR       D   11      mr22,   indonesia ru 9r     s
          piah
THB       D   12      mr44,   thai baht  8r     s
SGD       D   13      mr44,   singapore do 7r     s
          llar
TWD       D   14      mr44,   taiwan dolla 7r     s
          r

16 record(s) listed

```

Note that some of the descriptions have wrapped within their display field.

⁶ A field number of zero indicates we are referring to the item-id.

You probably understand that we have just created a set of descriptions of the data we are going to store in the database – something like an SQL schema – but it won't be totally clear what the elements of the definitions mean.

The 'D' in the first field indicates that this is a 'D-type' (direct) dictionary item. A 'D-type' item describes the date in the file.

An 'I' in this field would indicate that this field is an 'I-type' (indirect) field. An indirect field may also go by the name of a virtual field, a lookup field, or a calculated field in other databases.

The second field contains the field number of the data. The listing shows that we are going to store the US Dollar data values in the first field of the database. An 'I-type' dictionary item would contain an expression in this field.

A field number of zero indicates that we are referring to the ID field.

The third field contains a conversion code. This code is used to convert the data between an internal (storage) and an external (display) format. The conversion code we entered for 'IDR' (the Indonesian Rupiah) was 'MR22,.'. The 'MR' means that we are using a masked decimal conversion where the output should be right-justified. The first '2' means that we wish to display 2 decimal places, while the second '2' indicates the position of the implied decimal point in the data. The comma indicates that we should insert thousands separators in the output format.

Let's give an example. In January of 1999, there were 4,602.73 Indonesian Rupiah per NZ Dollar. If we apply an input conversion of 'MR22,.' to this value, we get an internal storage value of 460273. If we apply an output conversion of 'MR22,.' to the internal value of 460273 then we get a display value of 4,602.73.

We also used a conversion code of 'D' for the DATE dictionary item. 'D' is a generic date conversion. This can take on many variations as we will see later.

The fourth field is the display name for the field.

The fifth field gives the field width and justification.

The sixth field tells us whether the data is single or multi-valued. All this data is single valued.

We haven't used the seventh field. We can enter the name of an association in this field. An association links several fields together so the query processor knows to process them as linked fields.

Now we have a data set ready to be imported, and a basic set of dictionary items to describe the data. Let's do the import.

Import the data

The section above described the process of creating a database file and populating the data dictionary for the file. There still remains the task of populating the actual data from the csv file created earlier.

To simplify this process, a SDBasic program has been created, along with the required csv file. These items can be downloaded from the SD website https://stringdatabase.com/manual_examples.zip

Steps to use:

Download the zip file and extract.

Create a directory file in your account with the following command:

```
CREATE.FILE EXAMPLES_DATA DIRECTROY
```

Copy the file xrate.csv into the EXAMPLES_DATA folder

Copy the file CREATE.XRATES into the BP folder.

From the SD command line enter:

```
BASIC BP CREATE.XRATES
```

```
RUN BP CREATE.XRATES
```

Did it work?

Theoretically, we now have all that exchange rate data in the SD file named XRATES. Lets check that.

```
SORT XRATES DATE Page 1
XRATES.... Date.....
18266      03 JAN 2018
18267      04 JAN 2018
18268      05 JAN 2018
18271      08 JAN 2018
18272      09 JAN 2018
18273      10 JAN 2018
18274      11 JAN 2018
18275      12 JAN 2018
18278      15 JAN 2018
```

Well ... something is in the file – but what is it? The ID that is being displayed is nothing like the date we had as the ID column in the spreadsheet. The answer is that the date has been converted to a serial number.

Let's try a little more:

```

SORT XRATES DATE USD GBP AUD JPY EUR
XRATES.... Date..... us dollar uk pound  aus dollar  jap yen  euro...
18266      03 JAN 2018      0.7095  0.5219  0.9066  79.71  0.5883
18267      04 JAN 2018      0.7086  0.5245  0.9057  79.86  0.5901
18268      05 JAN 2018      0.7154  0.5275  0.9106  80.68  0.5923
18271      08 JAN 2018      0.7171  0.5283  0.9119  81.10  0.5954
18272      09 JAN 2018      0.7180  0.5292  0.9144  81.23  0.5998
18273      10 JAN 2018      0.7152  0.5285  0.9148  80.38  0.5992
18274      11 JAN 2018      0.7201  0.5330  0.9143  80.26  0.6022
18275      12 JAN 2018      0.7270  0.5362  0.9206  80.85  0.6030
18278      15 JAN 2018      0.7258  0.5284  0.9170  80.45  0.5953
18279      16 JAN 2018      0.7289  0.5284  0.9161  80.78  0.5944
```

If we check back to our source spreadsheet, we find that the data being displayed matches with the values recorded there.

These SORT commands are actually basic SDquery reports. We will develop more advanced reports in the later sections on Sdquery.

Add another file to the database

The *SDquery* section will cover (amongst other things) how to look up information in other files. To do this, we need another file.

Our previous file looked at exchange rates. Let's now create a file that contains interest rate data. We can get this data from the same site as the exchange rate data obtained earlier. See:

<https://www.rbnz.govt.nz/-/media/project/sites/rbnz/files/statistics/series/b/b2/hb2-daily.xlsx>. Download the spreadsheet marked 'B2 Daily (2018 to current)'.

This spreadsheet has some cells containing '-', which will be a nuisance for us. Select all the data in the spreadsheet, and use the 'Search and replace' function to replace all of these with nulls.

Delete rows 1 to 5, then go to the bottom of the spreadsheet and delete the comment rows there. Save the spreadsheet as a csv file.

Now let's create the *SD* file, and enter some dictionary items to define the data and the location of each element for the data import.

```
CREATE FILE IRATES
MODIFY DICT IRATES
```

Create the following dictionary items:

ID	Type	Loc	Conv	Name	Format	S/M	Assoc
DATE	D	0	D	Date	11R	S	
OCR	D	1	MR22,	Official Cash Rate	7R	S	
OVERNIGHT	D	2	MR22,	Overnight Cash Rate	7R	S	
DAYS30	D	3	MR22,	30 Day Bank Bill	7R	S	
DAYS60	D	4	MR22,	60 Day Bank Bill	7R	S	
DAYS90	D	5	MR22,	90 Day Bank Bill	7R	S	
YRS1	D	6	MR22,	1 Yr Govt Bonds	7R	S	
YRS2	D	7	MR22,	2 Yr Govt Bonds	7R	S	
YRS5	D	8	MR22,	5 Yr Govt Bonds	7R	S	
YRS10	D	9	MR22,	10 Yr Govt Bonds	7R	S	

While there are other columns in the spreadsheet, we won't import them.

The section above described the process of creating the second database file and populating the data dictionary for the file. There still remains the task of populating the actual data from the csv file created earlier. To simplify this process, a SDBasic program has been created, along with the required csv file. These items can be downloaded from the SD website

https://stringdatabase.com/manual_examples.zip

Steps to use:

Download the zip file and extract

Copy the file irates.csv into the EXAMPLES_DATA folder

Copy the program file CREATE.IRATES into the BP folder.

From the SD command line enter:

BASIC BP CREATE.IRATES

RUN BP CREATE.IRATES

Did it work?

Theoretically, we now have all that interest rate data in the *SD* file named IRATES. Lets check that.

sort IRATES DATE OVERNIGHT DAYS90 YRS10

The resulting list of interest rates should match the data in the source spreadsheet.

sort irates	date	overnight	days90	yrs10	Page 1				
IRATES....	Date.....	Overnight	Cash	Rate	90 Day	Bank	Bill	10 Yr	Govt Bonds
18266	03 JAN 2018			1.74			1.89		2.77
18267	04 JAN 2018			1.51			1.88		2.77
18268	05 JAN 2018			1.50			1.87		2.76
18271	08 JAN 2018			1.59			1.88		2.78
18272	09 JAN 2018			1.63			1.87		2.81
18273	10 JAN 2018			1.72			1.87		2.85
18274	11 JAN 2018			1.58			1.87		2.87
18275	12 JAN 2018			1.50			1.87		2.87
18278	15 JAN 2018			1.76			1.88		2.87
18279	16 JAN 2018			1.52			1.88		2.88
18280	17 JAN 2018			1.75			1.89		2.88
18281	18 JAN 2018			1.58			1.88		2.91
18282	19 JAN 2018			1.75			1.88		2.94
18285	22 JAN 2018			1.76			1.88		2.98

Add a third file to the database

The third file we are going to use is somewhat larger, and rather than stepping you through the creation of the file, you can just download it. There are a few supporting files too.

Unzip the files, and place them in your user account. Now, we'll need to add entries to the voc so that *SD* can find them. Use one of the editors (see the section on EDITORS) to create the following voc entries:

```
CT VOC NCY.C NCY.R TEX.H TEX.QCH
VOC NCY.C
1: F
2: NCY.C
3: NCY.C.DIC

VOC NCY.R
1: F
2: NCY.R
3: NCY.R.DIC

VOC TEX.H
1: F
2: TEX.H
3: TEX.H.DIC

VOC TEX.QCH
1: F
2: TEX.QCH
3: TEX.QCH.DIC
```

[An alternative approach is to create the files from the *SD* command prompt, then delete the files using a file manager. Finally, drop the downloaded files into the places where you deleted the original files].

We now have a set of files, and a matching set of file pointers.

These files contain the following data:

NCY.C	Country names
NCY.R	Region names
TEX.H	Harmonised code descriptions
TEX.QCH	New Zealand export data by quarter, country, and HS chapter.

We'll investigate this data later.

Editors

SD has three editors for use in a terminal emulation environment. These are:

ED A basic line editor
MODIFY A specialised editor used for editing dictionary items and data files
SED A full screen editor usually used for editing program source code

Although we have already used *ED* (when we created the *MASTER.LOGIN* item), this book will not use it again. Nor will it cover the use of *SED*. To fully understand these editors, see the on-line help file.

Of the *SD* editors, it is *MODIFY* that you should particularly learn.

When used to edit dictionary items, *MODIFY* presents you with appropriate prompts for each line of the dictionary item. Less obviously, *MODIFY* compiles dictionary items as they are filed, thereby highlighting syntactical errors before the dictionary item is used.

When used to modify a data item, *MODIFY* uses the file dictionary to create the prompts displayed to the user. In contrast, the *ED* editor simply displays a line number which is unhelpful if you do not know the position of each data element in the item.

The basic syntax for using the *MODIFY* editor is:

```
MODIFY {DICT} filename {list of item-id's}
```

If no item-id's are specified, then *MODIFY* will prompt you for an item-id, or if a select-list is present, then *MODIFY* will take the item-id's from the select-list (see Section Error: Reference source not found for information on select-lists). Example commands are:

```
MODIFY DICT XRATES  
MODIFY DICT XRATES USD  
MODIFY XRATES  
MODIFY XRATES 15346
```

In the first and third examples, *MODIFY* will prompt for an item-id, while in the other two examples, the item-id has been supplied on the command-line.

The command stack and command editing

SD is a command driven environment – that is, you type commands from the keyboard to control what *SD* does. *SD* stores these commands as you work, and provides quick access to the last 99 commands issued (this number is configurable), so you can either re-run an earlier command or edit the command before running it.

These editing facilities are particularly important for use with *SDQuery*. *SDQuery* commands are often built up over a series of iterations. This may start with a basic statement that selects and sorts records. Break points and output data will then be added to this, followed by headings and footings. The final command may cover several lines on the screen.

There are two sets of command line editing facilities. The first are the ‘dot’ commands that all are present in all multi-value environments (with minor differences in each environment). The second allows direct editing of commands on the command stack using the arrow keys.

The dot commands

The dot commands are literally a dot (period) followed by a single character. Some of these commands are further followed by parameters that control *SD*’s response to the command.

To see the list of dot commands, type `?.` (dot question-mark) at the command prompt:

```
?.
.An text      Append text to command stack entry n.
.Cn/s1/s2/G  Replace s1 by s2 in stack entry n. G = global replace.
.Dn          Delete stack entry n.
.D name      Delete named sentence or paragraph.
.In text     Insert text as stack entry n.
.Ln         Display last n lines from stack. Default is 20.
.L name      Display named sentence or paragraph.
.Rn         Recall stack entry n to top of stack.
.R name      Recall named sentence or paragraph to stack.
.S name s e  Save stack entries s to e as sentence or paragraph name.
.Un         Convert stack entry n to uppercase
.Xn         Execute command n. Default is 1.
.X file id   Execute command stored in named file and record
Spaces are required where shown. n defaults to one in all cases if omitted.
Use .DP, .LP, .RP and .SP to reference private VOC file.
```

The most common dot command you will use is `.L` (dot L). This provides a listing of recent commands:

```
.L
20 MODIFY DICT XRATES
19 SORT XRATES
18 MODIFY DICT XRATES
17 SORT XRATES
16 SORT XRATES DATE USD GBP AUD JPY EUR
15 FTTCL
14 FTSERVER 1
13 CLEAR.FILE DATA IRATES
12 sort dict irates
11 MODIFY DICT IRATES
10 SORT IRATES
09 FTTCL
08 FTSERVER 1
07 SORT IRATES DATE OVERNIGHT DAYS90 YRS10
06 CT IRATES 15346
05 sort xrates
04 CT XRATES 15346
03 sort irates
02 SORT IRATES DATE OVERNIGHT DAYS90 YRS10
01 SORT XRATES DATE USD GBP AUD JPY EUR
```

The most recent commands are at the bottom of the list nearest the current cursor position. The default number of commands is 20, but this can be changed by specifying a number in the `.L` command:

```
.L5
05 sort xrates
04 CT XRATES 15346
03 sort irates
02 SORT IRATES DATE OVERNIGHT DAYS90 YRS10
01 SORT XRATES DATE USD GBP AUD JPY EUR
```

To execute a command that is already on the command stack, use `.xn` where `n` is the number of the command. To repeat the sort on the dictionary of the `IRATES` file (command number 2), we would type `.x2`

```
.x2
SORT IRATES DATE OVERNIGHT DAYS90 YRS10
IRATES.... Date..... Overnight Cash Rate 90 Day Bank Bill 10 Yr Govt Bonds
15346 05 JAN 2010 2.52 2.80 6.13
15347 06 JAN 2010 2.37 2.79 6.07
15348 07 JAN 2010 2.38 2.78 6.06
15349 08 JAN 2010 2.25 2.78 6.06
```

When we execute a command like this, the specified command is executed as if it were just entered at the keyboard. This duplicates the existing command, and pushes all previous commands down the stack. In this case, we now have the command:

```
SORT IRATES DATE OVERNIGHT DAYS90 YRS10
```

at both position 1 and position 3 on the stack.

The `.c` command changes the text in a command. While we could change the text in any command on the stack, it is usually most convenient to operate on the first command. In this case, we may need to retrieve (`.R`) an earlier command from the stack.

```
.R2
02 SORT XRATES DATE USD GBP AUD JPY EUR
.C/AUD/CNY
01 SORT XRATES DATE USD GBP CNY JPY EUR
```

This sequence has retrieved the 2nd command from the stack, and then changed 'AUD' to 'CNY' in the list of currencies to display. We could then execute this new command simply by typing `.x`

```
.X
SORT XRATES DATE USD GBP CNY JPY EUR
XRATES.... Date..... US Dollar UK Pound Chinese Yuan Jap Yen Euro...
15346 05 JAN 2010 0.7344 0.4564 5.0145 67.98 0.5096
15347 06 JAN 2010 0.7343 0.4591 5.0129 67.39 0.5113
15348 07 JAN 2010 0.7378 0.4606 5.0376 68.11 0.5119
15349 08 JAN 2010 0.7325 0.4597 5.0014 68.50 0.5119
```

In this case, because we did not specify a command number, it assumed the first command.

Use of the `.S` command and the alternate form of the `.X` command is covered later in this book (see Section Error: Reference source not found). Otherwise, see the official *SD* documentation for information on the rest of the dot commands.

Editing keys

SD provides a much simpler and more intuitive way to edit the commands on the command stack than by using the dot commands. This is by using the editing keys (arrow keys, Home, End) on your keyboard.

Press the Up Arrow at the command prompt. The first command on the stack will be displayed at the prompt, with the cursor at the home position. Press the Up Arrow again, and the next command will be displayed, and so on through the command stack. Pressing the down arrow will bring you back down the command stack.

When you have got to the command you want, you can use the Left and Right Arrow keys to move through the command, and the Home and End keys to move to each end of the command. Typing text in from the keyboard will insert the text into the command at the cursor position, pressing Delete will delete the text underneath the cursor, and using the Backspace key will delete the text before the cursor.

When you have finished editing the command, press Enter to execute it.

If you want to abandon the editing you have made, use the Up or Down Arrow keys to move off the command, or use Ctrl-G to return to the command prompt. Your changes won't be saved.

There are other keyboard editing commands and configuration options. Search the help file for "command editor" for more information.

Preserving the command stack between sessions

Command stacks are saved between sessions in the `stacks` folder of your entry account. The `stacks` folder isn't visible from within *SD*, but you can see it using a file manager.

Now – what was that about the entry account? There is potentially a `stacks` folder in every account. This is created the first time that a command stack is saved in the account. So, if you can't see a `stacks` folder in the account, it is because no stacks have been saved there.

If you have multiple accounts and there is a `stacks` folder in each of them, then which stack is used when? Where is your current session stack actually saved?

You use the stack from the account by which you enter *SD*, and the stack is saved there when you exit – even if you have logged to another account during the session.

Once you have *SD* set up, you will usually log directly into your normal working account, and you won't notice that there are different stacks in other accounts (because you'll always use your account stack).

If your account is not saving the stack, create an item in the VOC named `$COMMAND.STACK` with an X in field 1. This is automatically added to new accounts when they are created, but may not be there for some old accounts (or it may have been deleted).

If you view the `stacks` folder from a file manager, you will find one stack in there for each user (that has entered that account). These are simply text lists which you can edit with a normal text editor. Note that you should be logged off from *SD* before you edit these items in this manner.

SD Database Files

Background

The section “Multi Value File Concepts” gave a brief overview of the files in the *SD* system, and also made passing reference to one of the advantages of the multi-value database file structures. This advantage was that the database was able to find any record in the database, no matter what the size of the database, with a single disk read – as long as the database was given the ID of the record. However, it didn’t really explain the mechanics of the process used to achieve this.

This section will look at how multi-value databases structure their files. This will explain how the database can find any record in the database, and look at other characteristics of the multi-value file structure.

The explanation will initially use the fixed-size files found in other multi-value databases because these illustrate the principles clearly. *SD* uses dynamic files, where the file size changes as data is added or deleted. The data storage principles are the same with dynamic files, but the database system does some extra work in the background when dynamic files are resized.

Note that this is a somewhat technical discussion. Readers can skip most of this section without missing any essential items from a practical usage perspective. However, most people should read Section File VOC Entries which deals with the voc entries for files.

Hashed Files

Traditional hashed files

Multi-value databases use what are termed ‘hashed files’ for data storage. Hashed files use the following principles:

- The file is made up of a series of storage containers (frames or buckets)
- The database “hashes” the ID of the record to generate a large number. This number is then divided by the number of buckets in the file (the modulo) and the remainder calculated.
- The record is then stored in the bucket indicated by the remainder.

Consider the following example:

A file has a base size of 7 frames. We want to store an item with an ID of 12345 in the file. The ID hashes to a value of 545,673⁷. Dividing this hash value by 7 and taking the remainder gives us a value of 2. This means that the item will be stored in the third bucket (because we start counting from zero)⁸.

The above procedure is always followed when the record is saved to the file. It will also be followed if the database is given the record ID and asked to get the record. However, it won't be followed if the file is being processed in sequential order – in that case, the system simply reads the records in the order in which they appear in the database.

Following this process a little further, we can see that each frame (bucket) in the file will contain multiple records. How do we know this? Well, records have to start in one of the buckets defined by the file size. If there are more records than buckets, then there have to be multiple records per bucket.

So what happens when the database system looks for a record in the bucket? Basically, it reads the whole bucket, and does a search through the contents of the bucket for the record being requested.

What happens when the bucket overflows? This will happen when the combined size of all the records assigned to the bucket exceeds the bucket size. In this case, the excess data is loaded into an overflow frame, and that frame gets linked back to the base frame.

An overflowed file imposes a performance penalty on the database system. It increases the number of disk reads to find a given record, and severely overflowed files face greater risks of corruption.

Traditionally, multi-value databases used fixed file sizes. Consequently, overflowed files could become a severe problem. Regular checks on file sizes were recommended to ensure that files did not become severely overflowed.

Oversized files could also be a problem. In that case, a file would have many empty frames and use excess disk space. More importantly, any process that required all data in the file would take longer than necessary because all the empty frames would need to be read as the system gathered the data.

There are other aspects of hashed files you should be aware of:

- Hashed files work best when the item-id's hash to an even distribution
- Hashed files work best where all records have similar sizes

7 Using the standard PICK hashing algorithm. *SD* probably uses a different algorithm. To find the PICK hashing algorithm, google the comp.databases.pick newsgroup for "hashing algorithm".

8 This is a bit easier to understand when traditional MV databases defined the file as starting at a given frame number (file base). The group that the item belonged to was then determined by adding the remainder derived in this example to the file base. Therefore, if the remainder from the division was zero, the item would be stored in the frame specified by file base.

- Hashed files work best when the average record size is only a small proportion of the bucket size (say, less than 20 per cent)
- Large records pose particular problems

Let's consider these:

A hashed file will (almost) always have some unused space. Continuing the example used above, the file has a modulo of 7. If each bucket is 2KB in size, then the total primary file size (i.e. excluding any overflow) is 14KB. Further, the primary file will always be this size regardless of the number of items in the file.

Ideally, we seek to size the file such that the records occupy most of the primary file space. There are two basic requirements to do this:

- Each bucket within the file should be allocated a similar number of records
- The records should be of a size that allows efficient utilization of the bucket

Say we had 140 records to go into our file of modulo 7. Ideally, we would want 20 records to go into each bucket. However, if the item-ids did not hash evenly, then we may have some buckets with 50 items, while others have only 5 or 10.

How do we know if a set of item-ids will hash evenly? In most cases, we don't know. We create a file, populate it, and then analyze the file to find its distribution of records. If there is excessive variability in the distribution of records, then we would test other file sizes using a test utility. Once we find a suitable file size, then we would resize the file to match the item-ids.

However, we do know that item-ids in a sequential series will always hash evenly. How do we know this? When the item-id is incrementing by one for each new item, then the hashing algorithm will effectively increment the bucket pointer by one for each new item. Therefore a sequential series of item-ids will produce an equal number of records going to each bucket⁹.

What about the utilisation of space within the bucket? Consider a file with a bucket size of 1KB (the default size for *SD*). If our average record size is 150 bytes, then we can fit six records (900 bytes) into each bucket before that bucket goes into overflow. If we try to fit a seventh record, then the total size of all records in the group¹⁰ (1,050 bytes) exceeds the bucket size. Therefore, an overflow bucket gets allocated to that group and we end up with 1,024 bytes used in primary file space, and 26 bytes in overflow.

9 See the section on "Analyzing a file" for a slight qualification of this.

10 A group consists of the base bucket in primary file space plus any associated buckets in overflow. Therefore, a group has a minimum size of one bucket, but may consist of multiple buckets.

What if the average item size was 350 bytes? Then we'll only get two items into the bucket before it goes into overflow. But two items is only 700 bytes, so the file will never be efficiently utilised. For this size record, we would be better using a bucket size of 2KB (5 records before overflow), or 4KB (11 records before overflow).

What if the average record size was 600 bytes? In the case of 1KB frames, then we'd only get one record per group before using overflow frames, and once again, the file space will never be efficiently used.

What if the record size was variable? Well, hopefully the pseudo-random allocation of records to buckets will largely cancel out the random variation in record size. In that case, each file group will have a similar size. The greater the number of records in each group, the more likely it becomes that each group will have a similar overall size¹¹ – provided that record sizes are evenly distributed, and the records hash evenly. However, if large items cluster together, then some groups will go into overflow while others remain relatively unutilised.

Finally, what if the record size was larger than the bucket size? In this case, the first item into a bucket will completely fill the primary file space and extend into overflow. If other records are subsequently allocated to the same bucket, then all of these records will be stored in overflow.

Consider also the outcome if the file does not hash evenly. In this case, some groups will be hugely overflowed, while other groups remain empty.

Therefore, there are two challenges when dealing with large record sizes. Firstly, you want to minimise the number of records being allocated to each bucket so that the number of records in overflow space is minimised. Secondly, you want to minimise the number of empty buckets. The difficulty is that minimising the number of empty buckets tends to come at the expense of putting more records into overflow.

Most multi-value databases have got around this large record problem by moving large items to their own special file space, and storing only a pointer to the large record in the primary file space. This has the advantage of ensuring the primary file space is used efficiently, but comes at the expense of requiring an additional disk read to read the record – the first disk read will retrieve the pointer from the primary file space, while the second will read the record itself.

Dynamic hashed files

SD uses dynamic hashed files. The 'dynamic' part of the name means that the files resize themselves automatically as records are added or deleted.

11 For example, Rocket Software recommends that *UniData* group sizes should be at least 10 times the average item size. Even after allowing for some unused space, this implies there should be at least 8 records per group, meaning that groups are more likely to be evenly sized than if there were only 2 items in the group.

Dynamic hashed files retain the key advantage of normal hashed files of single read access of a given record even without indexing, while removing (most of) the file administration procedures that accompany traditional hashed files.

A quick summary of how dynamic files work follows:

- *SD* continually calculates a load factor for each file. This is the total file size as a percentage of the primary file space.
- The load factor will change as records are added, modified, or deleted.
- If the load factor exceeds the `SPLIT.LOAD` parameter (default = 80%), then *SD* adds another bucket to the primary file space, and splits an existing bucket to provide records for the new bucket.
- If the load factor falls below the `MERGE.LOAD` parameter (default = 50%), then *SD* will merge some of the buckets.
- If a record is larger than the `LARGE.RECORD` parameter (default = 80% of bucket size) is encountered, then this record will be stored in the indirect record space, with only a pointer to this record stored in the primary file space.

In many cases, you can simply leave *SD* to manage file sizes for you. However, if you are seeking to maximise performance and/or you have unusual record sizes, then you may wish to change some of the default settings. Some of the possible settings have been alluded to above.

In addition to the `SPLIT.LOAD`, `MERGE.LOAD`, and `LARGE.RECORD` parameters, you may also want to change the `GROUP.SIZE` parameter. `GROUP.SIZE` sets the number of 1KB blocks that make up each bucket in the file. In some cases, you may also want to define a `MINIMUM.MODULUS` for the file.

There are a couple of questions which spring to mind regarding these parameters:

- How can we tell how the file is structured?
- How do we set or change these parameters?

Analyzing a file

SD provides a program to analyze a file to help you determine whether it is appropriately configured.

This program is `ANALYZE.FILE`:

```
ANALYZE.FILE filename {STATISTICS}
```

For example:

ANALYZE.FILE IRATES

```
Account      : /home/sd/user_accounts/sdintro
File name    : IRATES
Path name    : /home/sd/user_accounts/sdintro/IRATES

Type         : Dynamic, version 2
Group size   : 2 (2048 bytes)
Large record size : 1638
Minimum modulus : 1
Current modulus : 49
Load factors  : 80 (split), 50 (merge), 80 (current)
File size (bytes) : 145408 (102400 + 43008)
```

This starts with basic information about where the file is located in the host filesystem, and then provides information about the file configuration and status. In this case, the file is using a bucket size of 1 KB, a `LARGE.RECORD` size of 1638 bytes (80% of the bucket size), a `SPLIT.LOAD` of 80 per cent, and a `MERGE.LOAD` of 50 per cent. The current load factor is recorded as 80 per cent.

Other information notes that there are 49 buckets in the file at the moment (Current modulus), that the primary file space is 102,00 bytes, and the overflow file space is 43,008 bytes, for a total file space of 145,408 bytes. If you view the file through Ubuntu Files, you will find that these reported file sizes match the sizes of the %0 and %1 files respectively.

Adding the `STATISTICS` option provides additional information about the file:

ANALYSE.FILE IRATES STATISTICS

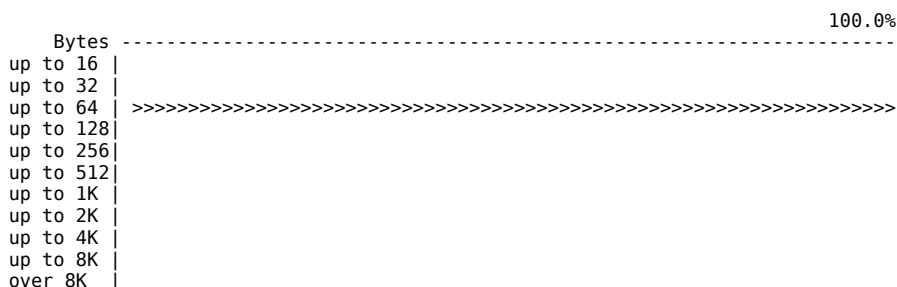
ccount : /home/sd/user_accounts/sdintro
File name : IRATES
Path name : /home/sd/user_accounts/sdintro/IRATES

Type : Dynamic, version 2
Group size : 2 (2048 bytes)
Large record size : 1638
Minimum modulus : 1
Current modulus : 49 (0 empty, 15 overflowed, 0 badly)
Load factors : 80 (split), 50 (merge), 80 (current)
File size (bytes) : 145408 (102400 + 43008), 80572 used
Total records : 1775 (1775 normal, 0 large)

	Per group:	Minimum	Maximum	Average
Group buffers	:	1	2	1.31
Total records	:	23	65	36.22
Used bytes	:	200	1580	1644.33

	Bytes per record:	Minimum	Maximum	Average
All records	:	36	48	45.39
Normal records	:	36	48	45.39
Large records	:			

Histogram of record lengths



Extra information included tells us that the file contains 1,775 records, which take up 80,572 bytes. Of the 49 buckets in the file, none are empty and 15 are overflowed.

The next section of information tells us a bit about the use of file space. The 'Group buffers' line tells us that each group in the file uses a minimum of one buffer (bucket), and a maximum of two buffer, for an average of one point three one buffer per group. If there were some empty buckets, then the minimum value would be reported as zero. Some buckets are overflowed, then the maximum value would be two (or more).

The 'Total records' line tells us that each group contains between 23 and 65 records, with an average of roughly 36 records per group. Likewise, the 'Used bytes' line tells us that each group contains between 200 and 1,580 bytes.

In the 'Bytes per record' section, we see that records occupy between 36 and 48 bytes each, at an overall average of 45 bytes per record. All of these records are "normal" - i.e. there are no large records.

Finally, the histogram shows that all the record sizes fit into a single size range.

This information is telling us that the file is fairly efficiently utilised. Similarly, records are of fairly even size.

Could we get better utilisation? Maybe. The smallest group is less than half full (200 bytes), while the largest group is going into overflow (1,580 bytes). There is a similar range in the number of records per group. This suggests that the ID's are not hashing evenly. You will recall from part one of this book that this file contained monthly interest rate indicators and used an ID of YYYYMM. Given that this ID is not random, and that the MM part of the ID will repeat over a short period (01 to 12), it is not surprising that the hashing is not even.

Setting or changing parameters

File parameters may be set at the time of file creation using the CREATE.FILE verb, or changed at some later time using the CONFIGURE.FILE verb.

Additionally, the default GROUP.SIZE parameter may be changed for the whole SD system by editing `/etc/config` or individually for specific users using the CONFIG command:

```
CONFIG GRPSIZE value
```

where *value* = 1, 2, 4, or 8 and represents the number of 1 KB blocks making a bucket in the dynamic file.

Note the different name given to GROUP.SIZE when set via the configuration options. To see the current setting for GRPSIZE (and all other configuration options), simply type:

```
CONFIG
```

To set the values of these configurable parameters at the time of file creation, use the options in the CREATE.FILE verb:

```
CREATE.FILE filename {GROUP.SIZE size} {LARGE.RECORD bytes} {SPLIT.LOAD pct} {MERGE.LOAD pct}
```

where: *size* = 1, 2, 4, or 8 representing the number of 1 KB block in each bucket
bytes = size of record before the record is stored indirectly
pct = load percentage before split or merge operations

For example:

```
CREATE.FILE SALES GROUP.SIZE 2 LARGE.RECORD 800
```

Reconfiguring an existing file uses similar parameters:

```
CONFIGURE.FILE filename {GROUP.SIZE size} {LARGE.RECORD bytes} {SPLIT.LOAD pct}  
{MERGE.LOAD pct}
```

Note that changing the group size of an existing file requires exclusive access to the file – i.e. other users will be denied access to the file while the file restructures itself – but changes to other parameters will take place in the background.

Both `CREATE.FILE` and `CONFIGURE.FILE` have other configurable parameters, most of which are not covered here. See the online help for more information on these parameters.

We can use a couple of these other options to resize the `IRATES` file analysed earlier. We want to see if the file will hash evenly if the modulus is exactly 64. We can set the file to this size using the following command:

```
CONFIGURE.FILE IRATES MINIMUM.MODULUS 64 IMMEDIATE
```

After running this command, the file analysis looks like:

ANALYSE.FILE IRATES STATISTICS

```
Account      : /home/sd/user_accounts/sdintro
File name    : IRATES
Path name    : /home/sd/user_accounts/sdintro/IRATES

Type        : Dynamic, version 2
Group size   : 2 (2048 bytes)
Large record size : 1638
Minimum modulus : 64
Current modulus : 64 (0 empty, 0 overflowed, 0 badly)
Load factors : 80 (split), 50 (merge), 61 (current)
File size (bytes) : 176128 (133120 + 43008), 80572 used
Total records : 1775 (1775 normal, 0 large)

Per group:
Group buffers : Minimum 1 Maximum 1 Average 1.00
Total records : 23 34 27.73
Used bytes : 1036 1580 1258.94

Bytes per record:
All records : Minimum 36 Maximum 48 Average 45.39
Normal records : 36 48 45.39
Large records :
```

This change has moved all of the records in overflow back into primary file space, and reduced the load factor to 61 per cent. It has also evened up the distribution of records (range 23 to 34 records per group, down from 23 to 64), but it is still not completely even.

Example file configuration

Let's consider an example where you might want to use a file configuration that is different from the *SD* default:

Let's say we want to store the daily transactions of a medium size supermarket. Because this file will be growing constantly during the day, we want to create the file with an initial size large enough to hold all the expected transactions for the day – this way *SD* won't have to devote resources to file resizing during the day.

The supermarket has 20 checkouts, and will have an average of 12 of them open throughout the day. It is open from 7.00 in the morning until 11.00 at night. Each checkout processes an average of 25

customers per hour. Each transaction has an average of 35 line items. What characteristics should we assign to the file?

First we need to calculate the average item size. From this, we will be able to set the group size, and calculate the total file size. Given those two pieces of information, we should be able to derive the minimum modulo to set for the file.

Each record will consist of:

Field	Size
ID	4Key
Date	5
Time	5
Checkout No	2
Operator ID	4
Payment Reference	16or 1
Loyalty card number	16or 0
Total Value (excl tax)	5
Tax	4
Sub-total	61or 30
Item reference	16
Quantity (number)	1 or
Quantity (weight)	4
Price	4
Sub-total	22average

The key for this file is simply a sequential transaction number. We can quickly calculate from the numbers given so far that we expect 4,800 transactions per day. On this basis, the ID will usually be a 4 digit number.

Our average item size will be:

Header	61
35 lines @ 22 bytes	770
Attribute marks	12
Value marks	140
Total	983bytes

and our total file size will be:

$$12 \text{ checkouts} \times 16 \text{ hours} \times 25 \text{ customers} \times 983 \text{ bytes} = 4,718,400 \text{ bytes}$$

How should we configure this file?

These records are nearly 1 KB each on average. If we want 10 records per group (or more), then the group size needs to be in excess of 10 KB. The nearest we can come to this is to use the 8 KB group size.

We can now calculate the modulo for the file:

$$4,718,400 \text{ bytes} / 8,096 \text{ bytes per group} / 75\% \text{ load factor} = 777$$

Or if you want to be a bit more aggressive in your file sizing, you could use a load factor of 80 per cent to derive a modulo of 728.

Therefore, our CREATE.FILE command would look like:

```
CREATE.FILE DS,DS20081028 GROUP.SIZE 8 MIMIMUM MODULUS 780
```

Note this assumes use of a shared dictionary named DS (daily sales).

The biggest problem with this file configuration is the inherent variability in the size of supermarket orders. There are two particular aspects to this variability – firstly, the daily variation in the amount of business transacted along with seasonal variations; and secondly, the variability of transaction sizes within any given day. Let's consider the variability in transaction size first:

Individual purchases may range from a single item to a whole trolley load of 100 items. This effectively means that our record size will vary from 59 bytes (single item purchased using cash and without a loyalty card) up to 1,959 bytes for a 100 item purchase. Clearly, this range of record sizes will create potential problems for the filing system.

Some options for dealing with this variability include:

- ignoring it
- using the LARGE.RECORD setting to move the large items into indirect file space
- normalising the file to move the transaction lines to a separate file.

Ignoring the issue may not be ideal, but it is the simplest option, and may work well enough.

Using the large record setting does not fix the problem either, but does move the largest records (say those over 1,000 bytes) out of primary file space. This will effectively reduce the variability of the record sizes – but will have the side effect of reducing the total amount of primary file space required. Therefore, the minimum modulus could be specified a little smaller than was calculated above.

The final solution of suggesting file normalisation is akin to heresy when applied to multi-value databases, but is actually a good solution to the issue of sizing this data. On the other hand, it may complicate reporting and programming. Further, because of the necessity of providing a key for each item in the transactions file, this will actually use more disk space. However, it is well worth considering.

In this case, the file structures would consist of a header file with records of 30 to 61 bytes (plus 8 bytes for field marks), and a transaction lines file with records of 22 bytes plus a key of about 8 bytes

(transaction number, separator, transaction line number, and field mark). Both files would consist of evenly sized records which should assist in achieving good file utilisation, and the records are all small enough to allow the default group size of 1 KB.

The file sizes would be:

12 checkouts x 16 hours x 25 customers x 60 bytes = 288,000 bytes (header)

12 checkouts x 16 hours x 25 customers x 35 lines x 30 bytes = 5,040,000 bytes
(transactions)

and modulus would be:

288,000 bytes / 1,024 bytes per group / 80% load factor = 351 (header)

5,040,000 bytes / 1,024 bytes per group / 80% load factor = 6,152 (transactions)

Accordingly, the file creation commands would be:

```
CREATE.FILE DS,DS20081028 MIMIMUM.MODULUS 350
```

```
CREATE.FILE DT,DT20081028 MIMIMUM.MODULUS 6150
```

Once again, this assumes the use of shared dictionaries named DS (daily sales) and DT (daily transactions).

Note the following points:

- total file size has increased (5,328,000 bytes c.f. 4,718,400 bytes)
- the modulo of the transactions file is much larger than the previously calculated modulo (6,152 c.f. 780)
- the group size for each file is much smaller than calculated earlier (1 KB c.f. 8 KB).

Essentially, this file structure uses many small groups rather than fewer large groups.

The “right” structure of the data files will depend on local considerations. The above example is intended to give some guidance as to how the different file configuration options work, and when to use them.

Variability in the number of transactions per day could be handled by a set of factors which rate a given day relative to a “normal” day. Mondays and Saturdays may be rated at 120 per cent of normal, while the remaining days are about 90 per cent. In the two weeks before Christmas, days are given an additional factor starting at 100 per cent of normal 14 days prior to Christmas, increasing to 300 per cent of normal for the last full day of shopping before Christmas.

Directory Files

Hashed files are good for storing records which are relatively homogeneous in terms of size. Ideally, the records shouldn't be too large either. Traditional multi-value databases cannot store binary items (such as images) in hashed files either, but *SD* does not have this limitation.

For data that doesn't meet these criteria, you should use directory files. These are simply operating system directories (folders) which are also defined in the voc of *SD* so that *SD* can access them.

Within multi-value environments, the most common usage of directory files is for storing programs. This allows the use of external editors to write and modify programs used in the multi-value environment.

Similarly, because directory files are accessible from outside the multi-value environment, such files are frequently used to exchange data between the multi-value environment and external systems.

However, there are some limitations associated with directory files that should be noted. In large part, these limitations are the flip-side of the benefits of the directory files:

- file performance is lower than for hashed files (sorting and selecting data)
- directory files cannot be indexed
- you cannot apply triggers to directory files.

The restrictions on indexing and triggers are obvious enough – if processes outside *SD* add, modify, or delete records in directory files, then *SD*'s indexing and triggers cannot keep track of those changes. Therefore, data would become inconsistent (because trigger processes have not been applied), and indexes would become out of date.

The lack of indexing partly explains the lower performance of directory files. However, at a more fundamental level, it is more difficult for *SD* to sort and process directory files.

In summary then, use directory files for:

- programs
- some large items
- transferring data between *SD* and other environments.

Data Storage

Variable length fields

So far, there has been almost no mention of one of the key advantages of multi-value databases – variable length fields (and by association, records). This provides several benefits:

- display restrictions do not limit the amount of data that can be entered into a field
- empty fields do not take up any space in the database
- changing field lengths do not require any restructuring of the database.

These benefits are best shown by contrasting the situation with a typical SQL database:

SQL databases use a schema that rigidly defines the data entered into the database¹². A data field that is defined as containing 20 characters can only take a maximum of 20 characters, regardless of the actual length of the data attribute. In contrast, you can enter any amount of data into a multi-value database field – even if the dictionary item describing that field specifies a lesser field width. This highlights the fact that multi-value database dictionary items are descriptive (rather than prescriptive). Of course, the way this “extra” data is displayed will depend on the formatting instructions in the dictionary item, but at least the data is available in the database.

Likewise, when an SQL database defines a field with a width of 20 characters, the database will always allocate 20 characters to that field – even if that field is empty. In contrast, a multi-value database will not allocate any space to an empty field (other than a field marker). This has potential to make a multi-value database smaller than an equivalent SQL database.

So what happens when the database design needs to change? For example, the size of a data field needs to increase. In an SQL database, you usually need to restructure the whole database to allocate more space to the field. In a multi-value database, nothing changes at the database level – the database can already take any amount of data (within reason) in any field.

To sum things up, variable length fields add a great deal of flexibility to database design. You can focus on entering, processing, and displaying the data rather than worrying about the size of the data field that is to be entered.

The flip side to this flexibility is that it is easy to start doing things in multi-value environments without thinking through the design issues properly.

String representation

The other unusual aspect of multi-value storage is that the data is all stored as an ASCII string – that is, using alphanumeric characters. The number 12348 is literally stored in the database as ‘12348’.

¹² Some modern SQL databases have removed some of these restrictions. For example, Oracle allows fields to be redefined without a complete restructure of the database, while MS Access uses variable width fields for text fields, making the field width restriction advisory rather than definitive.

Once again, compare this with the situation in other databases. Typically, those databases will store numbers in fields defined as having a numeric format. Importantly, those numbers are converted to hexadecimal before being stored. Therefore, 12348 is actually stored as '303C' - i.e. 4 bytes instead of 5 – but the 4 bytes may well be stored within an 8 byte field.

What is the benefit of this? There is no great advantage in storage space requirements – particularly with the large capacity hard disks that are common on all modern hardware. Multi-value environments generally use less space for any given storage requirement, but this is more due to the variable length fields than to differences in numeric storage methods. It does make the contents of the database human-readable – but looking directly at the contents of the database files (through a file utility) is not recommended.

The major advantage (or disadvantage depending on your viewpoint) is that you gain flexibility by treating everything as a string:

- you don't need to define the fields as strictly integer, real, or string
- you can store string data in fields that are mostly numeric, and vice-versa.

The first point simply means that you don't need to formally define your fields – but you should do so anyway. The whole point of a field is to group data of similar types.

The second point is also not recommended. Storing data of different types within a field makes analysis of that data difficult.

Overall then, multi-value environments store data as strings largely because there is no formal database schema that says that a particular field must be of a specific data type. While it does give some advantages in terms of reducing the overhead of database design, this isn't a step that you should skip. You should design your database with the intention of storing particular data types in specific fields.

However, there are some advantages when it comes to programming in the database environment. These advantages are that variables can be re-used with changing the data type, and that string and numeric values can be combined together without type conversion (see page Error: Reference source not found for an example of this). Once again, these advantages do not come without disadvantages – notably the danger of trying to perform numeric operations on alphabetic characters.

Internal Data Storage

While it may seem unnecessary to know how *SD* stores data internally, this knowledge can actually help you (as a developer) to handle the data better.

The way that data is stored follows on directly from the previous two points – data is stored as a string in variable length fields. *SD* uses a number of special delimiter characters to identify the boundaries of fields, values, sub-values, and records. By knowing this, you can use *SD*'s special string handling functions to extract data fields, and groups of fields.

The full list of delimiter characters, their normal ASCII values, and their internal tokens are shown below:

Mark	ASCII	Token	Representation
Item	255	@IM	
Attribute or Field	254	@AM or @FM	^
Value	253	@VM]
Sub-value	252	@SM or @SVM	\
Text	251	@TM	

While the ASCII values and tokens are functionally identical, it is recommended that you use the tokens in your programs. This is partly because the code is more readable, and partly because it allows the delimiter character to be changed in the future (to take advantage of Unicode character sets for example).

The representation column in the table shows how delimiter characters are often displayed in text. Note, this is different from the way that your terminal may display the character. A terminal will use the character associated with the character set that you are using.

Prior to the introduction of standard tokens, many multi-value BASIC programs used constants for the same purpose:

```
EQUATE AM TO CHAR(254)
EQUATE VM TO CHAR(253)
EQUATE SM TO CHAR(252)
```

The program would then use the constants defined above in the rest of the program.

Now let's consider what a stored record looks like, and how we might use that information to write more efficient programs. In general, a stored record will look like:

```
item-id^a1^a2-v1]a2-v2-sv1\a2-v2-sv2^a3^a4-v1]a4-v2]a4-v3 etc
```

where:

```
a = attribute
v = value
sv = sub-value
```

When the item is read into a variable, the item-id is stripped off, while the variable contains the rest of the string (from a1 onwards) as is.

Let's say the item has 16 attributes, and is stored in the variable *rec*. We want to create a new item consisting of attributes 11 through 16. We could do this as follows:

```
newrec = ''
FOR ii = 1 TO 6
  newrec<ii> = rec<ii + 10>
NEXT ii
```

or we could do it as:

```
newrec = FIELD(rec, @AM, 11, 6)
```

Both methods get attributes 11 through 16 and assign them to a new record, but the second method, which treats the entire record as a delimited string, is much more efficient.

Older code may use a technique like:

```
xpos = INDEX(rec, @AM, 10)
newrec = rec[xpos + 1, 9999]
```

Similarly, if we want our new record to have attributes 1 to 6 and 12 to 16, then we could:

```
newrec = ''
FOR ii = 1 TO 11
  IF ii GT 6 THEN
    jj = ii + 5
  END ELSE
    jj = ii
  END
  newrec<ii> = rec<jj>
NEXT ii
```

or:

```
dummy = FIELD(rec, @AM, 7, 5)
newrec = rec[1, COL1() - 1]:rec[COL2(), LEN(rec)]
```

or:

```
newrec = FIELD(rec, @AM, 1 6):@AM:FIELD(rec, @AM, 12, 5)
```

Once again, the second and third methods treat the existing record as a string, and use string extraction techniques to build the new record from the constituent parts of the existing record.

Binary Data

As noted above, *SD* can store binary data in hashed files. This is slightly unusual in multi-value databases because binary data is likely to contain some characters that match the system delimiters (ASCII 251 to 255).

Just because this is possible, it doesn't mean that it is always a good idea. Use your judgement – large binary items such as images or movie files probably shouldn't be stored in hashed files. A more appropriate arrangement for such items would be to rename the binary item to a sequential numeric name and store the item in a directory file. The hashed file should then contain the real file name and the path to the renamed item in the directory file (or the *SD* filename and itemname).

Even when you do store binary data in directory files, you need to be aware of how *SD* handles such items:

SD assumes that an item stored in a directory files is a text item, and that the appropriate representation of that item within the multi-value world is to replace the line-ends with field marks. Clearly, that is the wrong thing to do if the item is actually a binary item.

To prevent this happening, `MARK.MAPPING` should be turned OFF prior to reading or writing the binary item, and then turned back ON after the read/write.

For more information on this, see the topics ‘Directory Files’ and `MARK.MAPPING` in the online help.

File VOC Entries

Basic concepts

A key feature of multi-value databases is that most things are defined in the voc file. This is really how `SDQuery` works – all the elements of a `SDQuery` must be present either in the voc or in the dictionary of the file being queried.

This is true for files too. It is the file entry in the voc that lets each part of `SD` find the file to operate on. Let’s prove this:

In Part 1 of Getting Started in `SD`, we created a file named `IRATES`. The voc entry for this file looks like:

```
:CT VOC IRATES  
VOC IRATES  
1: F  
2: IRATES  
3: IRATES.DIC
```

The ‘F’ on line 1 tells `SD` that this is a file. It is only the first character that is relevant to `SD`, so you can add a description of the file on this line:

```
VOC IRATES  
1: File of Intrest Rates  
2: IRATES  
3: IRATES.DIC
```

This description will be displayed when you list the files in the account using the `LISTF` or `LISTFL` commands.

Line 2 of the voc entry tells `SD` the name of the data file, while line 3 has the name of the dictionary file.

`SDQuery` uses the information in this item to find both the dictionary and the data file so that it can report on the file:

```

SORT IRATES WITH YEAR = "2020" DATE DAYS30 DAYS60 DAYS90 ID.SUP
Date..... 30 Day Bank Bill 60 Day Bank Bill 90 Day Bank Bill
03 JAN 2020          1.19          1.24          1.29
06 JAN 2020          1.20          1.24          1.28
07 JAN 2020          1.20          1.23          1.27
08 JAN 2020          1.20          1.22          1.24
09 JAN 2020          1.20          1.21          1.23

```

Now, let's copy the VOC entry to a temporary item, and delete the main VOC entry:

```

COPY FROM VOC IRATES,TEMPIRA

```

```

1 record(s) copied.

```

```

CT VOC TEMPIRA

```

```

VOC TEMPIRA

```

```

1: F

```

```

2: IRATES

```

```

3: IRATES.DIC

```

```

DELETE VOC IRATE

```

```

1 record(s) deleted

```

Note that although we've deleted the voc entry for the file, we haven't deleted the file itself. You can check this by looking at the SDINTRO account using *Ubuntu Files*. You can also display the contents of the file by using the TEMPIRA file pointer that we created:

```

SORT TEMPIRA WITH YEAR = "2020" DATE DAYS30 DAYS60 DAYS90 ID.SUP
Page 1
Date..... 30 Day Bank Bill 60 Day Bank Bill 90 Day Bank Bill
03 JAN 2020          1.19          1.24          1.29
06 JAN 2020          1.20          1.24          1.28
07 JAN 2020          1.20          1.23          1.27
08 JAN 2020          1.20          1.22          1.24
09 JAN 2020          1.20          1.21          1.23

```

But, if we try to reference the file using the FX.DAILY name, SDQuery will not be able to find the file:

```

SORT IRATES WITH YEAR = "2020" DATE DAYS30 DAYS60 DAYS90 ID.SUP

```

```

File not found

```

OK – Now let's put things back the way they should be:

```

COPY FROM VOC TEMPIRA,IRATES

```

```

1 record(s) copied.

```

```

CT VOC IRATES

```

```

VOC IRATES

```

```

1: F

```

```

2: IRATES

```

```

3: IRATES.DIC

```

```

DELETE VOC TEMPIRA

```

```

1 record(s) deleted

```

So, what have learnt from this:

- SD relies on voc entries to find files
- voc entries exist independently of the data and dictionary files, and can have a different name
- Files can exist without voc entries and vice-versa

- A file can have more than one voc entry at any given point in time
- We can manually create voc entries for existing files.

Different types of VOC entries

Dynamic and directory files

Both dynamic files and directory files have the same type of voc entry. We can see this by comparing the voc entry for FX.DAILY (a dynamic file) with that of the QUERIES file (a directory file), also discussed in an earlier section.

```
:CT VOC QUERIES
VOC QUERIES
1: F
2: QUERIES
3: QUERIES.DIC
```

Multi-files

Multi-files have a slightly different voc entry:

```
CREATE.FILE DICT TEMP
Created DICT part as TEMP.DIC
Added default '@ID' record to dictionary
```

```
CT VOC TEMP
VOC TEMP
1: F
2:
3: TEMP.DIC
```

```
CREATE.FILE DATA TEMP,TEMP1
Created DATA part as TEMP/TEMP1
```

```
CT VOC TEMP
VOC TEMP
1: F
2: TEMP/TEMP1
3: TEMP.DIC
4: TEMP1
```

```
CREATE.FILE DATA TEMP,TEMP2
Created DATA part as TEMP\TEMP2
```

```
CT VOC TEMP
VOC TEMP
1: F
2: TEMP/TEMP1@TEMP/TEMP2
3: TEMP.DIC
4: TEMP1@TEMP2
```

Note the following points from this sequence of commands:

- Following the initial dictionary creation, the voc entry does not contain any reference to a data file

- When a multi-file data portion is added to the file, the format of the second attribute changes, and a new attribute (attribute 4) is added:
 - Attribute 2 now contains the path name (relative to the account base) to the data file(s)
 - Attribute 4 contains the sub-file name of the multi-file
- As more data portions are added to the file, attributes 2 and 4 become multi-valued to store the path and file names.

The documentation notes that F-type VOC entries also have a fifth attribute. This controls the way the ACCOUNT.SAVE and FILE.SAVE commands treat this file. See the documentation for more details on this attribute.

Q-Pointers

Q-pointers are an alternate method of referencing a file. They are typically used when:

- the file resides in another account
- the file is part of a multi-file, and you wish to use a simpler filename
- the file has a long filename and you wish to use a simpler filename.

Q-pointers have the following structure:

```
1: Q
2: Account name
3: File name
```

To reference the BP file in the SDSYS account, the Q-pointer will look like:

```
1: Q
2: SDSYS
3: BP
```

We might name this item: BP.SDSYS

We would create this item using one of the editors available in *SD* (ED, SED, or MODIFY; MICRO if you ; or any other editor that you have enabled). Don't include the line numbers when entering the item! We could also use the SET.FILE command to create the Q-pointer.

A Q-pointer to one of the data portions of the multi-file created on the previous page would look like:

```
1: Q
2: SDINTRO
3: TEMP,TEMP2
```

If we called this item T2, then we could use the filename T2 in our SDQuery sentences, or within BASIC programs, whenever we wanted to access the data in TEMP,TEMP2.

Note that Q-pointers can be chained. That is, one Q-pointer can point to another Q-pointer, which in turn could point to another Q-pointer, or could point to an F-type entry. This method is not recommended but, it can be quite useful.

Consider the situation where you wish to move an important file from one account to a new account. You already have a number of accounts pointing to the file using Q-pointers. Rather than change all of the existing Q-pointers (and risk missing one of them), simply move the file to its new location, and place a Q-pointer in the voc of the old account pointing to the new location. When one of the other accounts wish to access the file, they will look up their own Q-pointer which will direct them to the old location. They will find a Q-pointer there which will direct them to the new location.

Manual creation of F-type VOC entries

It is clear that Q-type voc entries are created independently of the file itself. On the other hand, F-type voc entries are maintained automatically by *SD* when we create and delete files.

What is less apparent is that we can create F-type entries manually to point to existing files on the system. Such files may be within the current account, or have a totally different path.

For example, we created a Q-pointer to the `BP` file in the `SDSYS` account earlier. We could have defined that as an F-type record as follows:

```
1: F
2: /usr/local/sdsys/BP
```

or:

```
1: F
2: @SDSYS/BP
```

This is a slightly unusual example because we haven't created a third line in the voc entry for the dictionary path. The reason for this is quite simple – the `BP` file in `SDSYS` does not have a dictionary, and if we specify a path for it, *SDQuery* will report that it cannot open the dictionary. However, we would normally specify the path to the dictionary in attribute 3 of the voc entry.

The second example uses the predefined token `@SDSYS`. It is preferable to use this token than specify the pathname for `SDSYS` because some administrators will place *SD* in a non-standard location. By using a token for the location of the `SDSYS` account, we can transfer our applications between systems with confidence that they will still work in the new system.

SD automatically defines two more tokens to be used in this manner - `@TMP` and `@HOME`. See the documentation for further information on these tokens.

We could also create an F-type entry for a directory on the system so that *SD* can access the directory as if it were a file created by *SD*. For example:

```
1: F
2: /tmp
```

Once again, we haven't included a reference to a dictionary file because the 'Temp' directory doesn't have a dictionary.

Given that we can manually create F-type voc entries for existing files, and such entries serve a similar purpose as Q-pointers, when should we create F-type voc entries for existing files, and when should we create Q-type voc entries?

As with many issues in multi-value databases, the answer is one of principle, rather than what is enforced by the database. The database lets you use either type of voc entry, but what is the principle behind each type of entry?

An F-type entry is created when *SD* creates a data file. The file is created within the account, and the F-type voc entry represents an ownership of that file by the account.

A Q-type entry can point to anywhere – the current *SD* account, another *SD* account, or directory file elsewhere within the operating system. It does not confer ownership of the file – rather it simply says that this account uses that file.

Using these basic principles:

- Let *SD* manage the F-type entries for files within an account
- Use a manually created F-type entry to create a reference to a file that is not part of the *SD* file system
- Only have one F-type entry per file used by *SD*
- Use Q-type entries to refer to *SD* files in other accounts, or to provide files with simpler names (in any account).

Alternate Key Indices

An alternate key index is a means of increasing the speed of accessing a set of records from a file. It does this by indexing fields from the file, or expressions based on the file data, in a special lookup file. When you select on the main file using one of the indexed fields, *SD* can access the indexed data and return the list of matching records without having to search the entire main file.

Indexing becomes more important as:

- the main file gets larger
- the number of records to be selected becomes a small proportion of the total number of records in the file
- the load on the system becomes higher
- the need for rapid response times increases.

Let's see what performance gains we can achieve by indexing a file:

We'll use a file named PRODUCT-DATA. This file currently contains 2.14 million records. ANALYSE.FILE produces the following statistics:

```

Type           : Dynamic, version 2
Group size     : 2 (2048 bytes)
Large record size : 1638
Minimum modulus : 1
Current modulus : 59572 (0 empty, 8871 overflowed, 81 badly)
Load factors   : 80 (split), 50 (merge), 80 (current)
File size (bytes) : 153077760 (122005504 + 31072256), 98821548 used
Total records  : 2140774 (2140774 normal, 0 large)

      Per group: Minimum   Maximum   Average
Group buffers :           1           3       1.15
Total records :           10          99       35.94
Used bytes    :           36         2048    1658.86

      Bytes per record: Minimum   Maximum   Average
All records   :           24         116       46.16
Normal records :           24         116       46.16
Large records :

```

We can see that the modulo is nearly 60,000 groups, and that the file has some groups in overflow. Primary file space is around 120 MB, and there is 30 MB of overflow space.

The file has a key of:

```
yyyywwttnnnlll
```

where: yyyy = season

ww = week

ttnnn = plant identifier

lll = product line identifier

Typical selection processes involve one or more of these key components. Dictionary items to extract these components are:

Dictname	Type	Expression	Conv	Name	Format	S/M
SEASON	I	@ID[1,4]		Season	6R	S
WEEK	I	@ID[5,2]		Week	4R	S
ME.NO	I	@ID[7,5]		ME No	6L	S
LINE3	I	@ID[12,3]		Line	3R	S

We'll create a simple paragraph to time a selection from this file:

CT VOC INDEX.TEST

VOC INDEX.TEST

1: PA

2: TIME

3: SSELECT PRODUCT-DATA WITH ME.NO EQ "ME078" AND WITH SEASON EQ "2007"

4: CLEARSELECT

5: TIME

Running this (twice) on a freshly rebooted system gave the following results:

INDEX.TEST

```
21:41:34 4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
21:42:46 4 FEB 2010
```

INDEX.TEST

```
21:42:55 4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
21:43:46 4 FEB 2010
```

On the first run, the selection took 1m 12s. This time decreased to 51s on the second run. This decrease in time is due to a portion of the file remaining in memory after the first run.

Now, we want to create the indices. As we are using the *ME.NO* and *SEASON* dictionary items, we will create indices on these fields:

CREATE.INDEX PRODUCT-DATA SEASON ME.NO

```
Added index for SEASON
Added index for ME.NO
```

BUILD.INDEX PRODUCT-DATA ALL

```
Building index 'SEASON'...
2140774 records processed
Populating index...
Building index 'ME.NO'...
2140774 records processed
Populating index...
```

We could have used the *MAKE.INDEX* command rather than the combination of *CREATE.INDEX* and *BUILD.INDEX*.

We can use the *LIST.INDEX* command to view information about the indices:

LIST.INDEX PRODUCT-DATA ALL STATISTICS

```
Alternate key indices for file PRODUCT-DATA
Number of indices = 2
```

Index name.....	En	Tp	Nulls	SM	Fmt	NC	Field/Expression			
SEASON	Y	I	Yes	S	R	N	@ID[1,4]			
Index entries			Key values		Min Recs		Avg Recs		Max Recs	
2140774			12		147286		178397.8333		219143	
Index name.....	En	Tp	Nulls	SM	Fmt	NC	Field/Expression			
ME.NO	Y	I	Yes	S	L	N	@ID[7,5]			
Index entries			Key values		Min Recs		Avg Recs		Max Recs	
2140774			108		1		19821.9815		105665	

After restarting the system, the test is run again:

INDEX.TEST

```
22:05:24 4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
22:06:10 4 FEB 2010
```

INDEX . TEST

```
22:06:26 4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
22:06:30 4 FEB 2010
```

This time, the first selection took 46s, while the second selection took just 4s. Subsequent restarts confirmed this pattern.

In practice, selection times will be between these two values – depending on how frequently the PRODUCT-DATA file is used. The more frequently it is used, the greater the probability that the index will be in memory and the shorter the selection times.

SDQuery will automatically use any indices that are available. You can also utilise the indices from SDBasic (see the online help for more information on this).

Summary

This chapter has covered a wide range of topics. You should understand by now:

- *SD* uses two types of files – dynamic hashed files and directory files
- the basic principles of hashed files
- what makes the dynamic files used by *SD* different from static hashed files used by traditional multi-value databases
- how to analyse the dynamic hashed files
- how to configure the dynamic hashed files for improved performance
- the way that *SD* stores data on disk
- the different types of voc entry relating to files, how to create them manually, and when to use each type
- how to create alternate key indices on a file.

SDQuery

SDQuery is an ad-hoc reporting language that uses the definitions stored in the dictionary files to report on the data. The language contains elements for:

- data selection
- data sorting
- data grouping
- carrying out summary operations (totals, averages, percentages)

- formatting of data on output
- generating headers and footers on each page of output
- panning and scrolling of output when viewed on a monitor
- redirection of output to printers, text files, or to delimited files
- printer formatting (PCL printers only)

Most of these elements are optional. Therefore, you can start with a simple statement and then gradually add to it as you learn more of the language. This makes *SDQuery* relatively simple to learn and use.

SDQuery is intimately associated with SD dictionaries. Therefore, you can't really learn *SDQuery* without gaining a good understanding of what dictionary items do and how they are constructed.

This section aims to work through the key elements of *SDQuery* while introducing dictionaries to you. By the end of the section, you should be able to write comprehensive *SDQuery* statements and write dictionary items to use with those *SDQuery* statements.

Anatomy of a *SDQuery* Statement

General syntax

SDQuery statements always follow a general syntactical form. This is:

```
verb {DICT} filename {USING {DICT} filename} {selection.clause } {sort.clause}
{display.clause} {record.id...} {FROM select.list.no} {TO select.list.no}
```

Given that most of the elements are optional, the simplest form of a *SDQuery* statement is simply:

```
verb filename
```

You have already used some of these commands, such as:

```
SORT VOC
SORT XRATES
SORT DICT XRATES
```

The most common verbs are `LIST` and `SORT`. Despite the difference in name, both verbs will sort the data. However, the `LIST` verb will only do this when a sort clause is included in the statement, while the `SORT` verb will always sort the data. The key difference is that the `SORT` verb appends a final sort by the item-id. Therefore, the command:

```
SORT INVOICES BY INV.DATE
```

is equivalent to:

```
LIST INVOICES BY INV.DATE BY @ID
```

In most of this section, we will only use the SORT verb. Some other verbs will be introduced later in the section (notably SELECT, SSELECT, and SEARCH), but for others you should refer to the *SD* documentation. These verbs are also listed in the Quick Reference at the back of this book.

Selection clause

As its name implies, the selection clause restricts the set of items to be reported on to those matching one or more criteria.

The general format of the selection clause is:

```
WITH {EVERY} condition {AND | OR condition}
```

where:

condition is: *field operator value*

or: *field1 operator field2*

and operator is one of the terms in the following table:

Operator	Synonym	Synonym	Synonym	Synonym
EQ	=	EQUAL		
NE	#	NOT	<>	><
LT	<	LESS	BEFORE	
LE	<=	=<		
GT	>	GREATER	AFTER	
GE	>=	=>		
LIKE	MATCHES	MATCHING		
UNLIKE	NOT.MATCHING			
SAID	SPOKEN	~		
NO				
BETWEEN				

Note that some operators have multiple synonyms. *SD* does not care which synonym you use – it offers you a choice for your convenience.

Creating a dictionary item for use in selecting data

Say we want to display the exchange rates for the year 2018. We could write a *SDQuery* statement like:

```
SORT XRATES WITH YEAR EQ 2018
```

Traditionally, multi-value databases have required that the comparison value in the expression be enclosed in quotes. So, in other databases, we would need to write:

```
SORT XRATES WITH YEAR EQ "2018"
```

SD makes these quotes optional. You can choose to include them or omit as you please. But if you are going to use one of the other multi-value databases, it would be good practice to quote your comparison values.

Running this commad, *SD* responds with:

```
YEAR is not a field name or expression
```

This means that *SD* has not been able to find a definition of the word *YEAR* – it does not appear in either the file dictionary or the voc of the account. Therefore, we need to define the word *YEAR* so that *SDQuery* understands what we mean.

To define *YEAR*, we use the *MODIFY* editor.

MODIFY DICT XRATES

Type in *YEAR* at the ID prompt, and define the item as follows:

ID	Type	Loc	Conv	Name	Format	S/M	Assoc
YEAR	I	OCONV(@ID, 'DY')		Year	4R	S	

Once you have filed the item, exit from the *MODIFY* editor and retry the *SDQuery* statement:

```
XRATES....
18266
18267
18268
18271
18272
18273
18274
18275
18278
```

Let's add the date so that we can see if we really have 2018 information:

```
SORT XRATES WITH YEAR EQ "2018" DATE
XRATES.... Date.....
18266      03 JAN 2018
18267      04 JAN 2018
18268      05 JAN 2018
18271      08 JAN 2018
18272      09 JAN 2018
18273      10 JAN 2018
18274      11 JAN 2018
18275      12 JAN 2018
18278      15 JAN 2018
18279      16 JAN 2018
```

That looks good. Let's see how we did that:

In the 'Type' field, we entered 'I'. 'I' stands for indirect, and it means that this is a calculated field. In *SD* jargon, this type of dictionary item is known as an I-type.

In the 'Loc' field, we entered the expression used to calculate the year. This expression was:

```
OCONV(@ID, 'DY')
```

OCONV is a function from the *SDBasic* programming language. It is the output conversion function. We have already come across conversions – those are the expressions that go into the *CONV* field of dictionary items.

In this case, the expression says apply an output conversion of 'DY' to the @ID field. You will recall that a 'D' conversion is a generic data conversion. In fact, any conversion code starting with a 'D' is a date conversion, and 'DY' means return the year of the passed (internal) date. We can see from the dates displayed that all have a year of 2018.

If all we have done is apply an output conversion, why didn't we do this by specifying a 'DY' conversion in the *CONV* field? Well, let's try that and see what happens.

Create a dictionary item *YEARX* as follows:

ID	Type	Loc	Conv	Name	Format	S/M	Assoc
YEARX	D	0	DY	Year	4R	S	

Let's see what output we get from it:

```

SORT XRATES DATE YEAR YEARX
XRATES.... Date..... Year YEAR
18266      03 JAN 2018 2018 2018
18267      04 JAN 2018 2018 2018
18268      05 JAN 2018 2018 2018
18271      08 JAN 2018 2018 2018
18272      09 JAN 2018 2018 2018
18273      10 JAN 2018 2018 2018
18274      11 JAN 2018 2018 2018
18275      12 JAN 2018 2018 2018

```

So, it displays the year correctly, just like our I-type item did. What about selecting data?

```

SORT XRATES WITH YEARX EQ "2018" DATE YEAR YEARX
0 record(s) listed

```

It didn't select any data. Why not?

Let's think about this. An output conversion in the CONV field is applied just before the data is displayed. However, when we are selecting and sorting the data, we are doing so in its internal data format – so it is still a date. Let's check that:

```

SORT XRATES WITH YEARX GE "01 JAN 2018" DATE YEAR YEARX
XRATES.... Date..... Year YEAR
18266      03 JAN 2018 2018 2018
18267      04 JAN 2018 2018 2018
18268      05 JAN 2018 2018 2018
18271      08 JAN 2018 2018 2018
18272      09 JAN 2018 2018 2018
18273      10 JAN 2018 2018 2018
18274      11 JAN 2018 2018 2018
18275      12 JAN 2018 2018 2018
18278      15 JAN 2018 2018 2018

```

So, even though it is displaying a year value, its internal value is still an entire date. So we can select on it as a date, but not as a year. Let's delete that dictionary item:

```

DELETE DICT XRATES YEARX
1 record(s) deleted

```

Let's add some other date type dictionary items:

```

MODIFY DICT XRATES
ID      Type  Loc              Conv  Name      Format  S/M  Assoc
MTHNO   I      OCONV(@ID, 'DM')      Month 2R    S
MTH     I      OCONV(@ID, 'DMA[3]')  MCT   Mth      3L    S
MONTH   I      OCONV(@ID, 'DMA')     MCT   Month    10L   S
DOM     I      OCONV(@ID, 'DD')      Day   3R      S
DOW     I      OCONV(@ID, 'DW')      Day   3R      S
DAY     I      OCONV(@ID, 'DWA[3]')  MCT   Day      3L    S

```

What does all this mean? Well, let's start by seeing what output they generate:

```

sort xrates date mthno mth month dom dow day sample 5
XRATES.... Date..... Month Mth Month..... Day Day Day
18477      02 AUG 2018 08 Aug August 02 4 Thu
18502      27 AUG 2018 08 Aug August 27 1 Mon
18604      07 DEC 2018 12 Dec December 07 5 Fri
18890      19 SEP 2019 09 Sep September 19 4 Thu
19049      25 FEB 2020 02 Feb February 25 2 Tue

```

MTHNO returns the month number of the year. It does this by applying a 'DM' conversion to the @ID (date) field.

MONTH returns the full name of the month, while MTH returns an abbreviated month name. In both cases, the central bit of the conversion is 'DMA' where the 'A' means return an alphabetic value. The MTH dictionary item then returns only 3 characters of this value.

DOM returns the day of month by applying a 'DD' conversion to the date, while DOW returns the day of week by applying a 'DW' conversion. The DAY dictionary item returns the day name by using a 'DWA' conversion.

Note that the dictionary items that return alphabetic values have a further conversion in the CONV field. This conversion is 'MCT' which capitalises the first letter of every word with the rest of each word in lower case. In recent versions of SD, we could also have used 'MCS' which capitalises the first word of a sentence, with the rest of the sentence in lower case.

Note that while these secondary conversions make the output look better than the default all-capitals values, they make selection by these dictionary items more difficult.

Selection by month number:

```

sort xrates with mthno eq "5" date mthno mth month dom dow day sample 5
XRATES.... Date..... Month Mth Month..... Day Day Day
19126      12 MAY 2020    05 May May          12  2  Tue
19486      07 MAY 2021    05 May May          07  5  Fri
19490      11 MAY 2021    05 May May          11  2  Tue
19498      19 MAY 2021    05 May May          19  3  Wed
20583      08 MAY 2024    05 May May          08  3  Wed

```

Sample of 5 record(s) listed

Note that we didn't specify the leading zero on the month number. This could fail in some other multi-value implementations as they could require that the leading zero be present in the comparison value.

Selection by abbreviated month:

```

SORT XRATES WITH MTH EQ "May" DATE MTHNO MTH MONTH DOM DOW DAY SAMPLE 5
Sample of 0 record(s) listed

```

This will not return anything regardless of the capitalisation of "May". If we take the 'MCT' conversion out of the dictionary item, then the following statement works:

```

SORT XRATES WITH MTH EQ "MAY" DATE MTHNO MTH MONTH DOM DOW DAY SAMPLE 5
XRATES.... Date..... Month Mth Month..... Day Day Day
19126      12 MAY 2020    05 MAY May          12  2  Tue
19486      07 MAY 2021    05 MAY May          07  5  Fri
19490      11 MAY 2021    05 MAY May          11  2  Tue
19498      19 MAY 2021    05 MAY May          19  3  Wed
20583      08 MAY 2024    05 MAY May          08  3  Wed

```

With the 'MCT' conversion in place, we need to do a case-insensitive comparison:

```

SORT XRATES WITH MTH EQ NO.CASE "MAY" DATE MTHNO MTH MONTH DOM DOW DAY SAMPLE 5
XRATES.... Date..... Month Mth Month..... Day Day Day
19126      12 MAY 2020    05 May May          12  2  Tue
19486      07 MAY 2021    05 May May          07  5  Fri
19490      11 MAY 2021    05 May May          11  2  Tue
19498      19 MAY 2021    05 May May          19  3  Wed
20583      08 MAY 2024    05 May May          08  3  Wed

```

Sample of 5 record(s) listed

In practice, we probably aren't going to make a selection on a literal month name – we would tend to use the month number for that. But we might want to select on someone's name, and we need to be aware of the impact of conversion codes on the selection process.

A full list of conversion codes can be found in the *SD* documentation, but some alternatives and their output is shown below for date value 16607 (Wednesday, 19 June 2013).

```

Code      Output
'D'       19 JUN 2013
'D2'      19 JUN 13
'D2/'     19/06/13          (or 06/19/13)
'D4/'     19/06/2013        (or 06/19/2013)
'D-YMD'   2013-06-19
'DW'      3
'DWA'     WEDNESDAY
'DWAL'    Wednesday
'DMA'     JUNE
'DMAL'    June
'DMAL[3]' Jun

```

You will realise by now that the text you enter in the 'Name' field of the dictionary is what appears in the column heading. Similarly, what you enter in the format field defines the width of the column – with some exceptions.

The MTHNO dictionary item specifies a format of 2R which means right-justify the field with a field width of 2 characters. However, the actual output displays the column heading fully, meaning that the actual field width is 5 characters wide.

Both the format codes and the conversion codes contain many options allowing powerful formatting of output from dictionaries. We'll cover some more of these later, but you need to read the manuals and help files to gain a full picture of their capabilities.

Multiple selection criteria

Often, we need to select on more than one criteria. For example, we want to display all dates in 2018 when the exchange rate has been below 65 US cents to the NZ dollar.

```

sort xrates with year eq "2018" and with usd lt "0.6500" date usd
XRATES.... Date..... us dollar
18540      04 OCT 2018    0.6498
18541      05 OCT 2018    0.6481
18544      08 OCT 2018    0.6427
18545      09 OCT 2018    0.6449
18546      10 OCT 2018    0.6491
18547      11 OCT 2018    0.6467

```

6 record(s) listed

Note that *SD* recognises the statement even though everything is in lower case.

In this statement, the `WITH` keyword is specified a second time after the `AND`. In *SD*, this is not strictly necessary – but other multi-value environments require the second `WITH` to be included.

It is generally a good idea to make your statements as compatible with other multi-value environments as possible. This is because you might have to work on one of these other systems, and if you aren't familiar with their syntax, you will find their query language to be quite restrictive.

In practical usage, you can have as many selection criteria as you like. However, if you have a lot of selection criteria, you may find it better to use multiple `SELECT` statements (see Section Multiple selection criteria) for selecting the data, followed by `LIST` or `SORT` statements to display the data.

There is another form of selection, where two criteria are applied to the same element. For example, display dates where the exchange rate is between 65 and 651 US cents:

```
SORT XRATES WITH USD GE "0.65" AND LE "0.651" DATE USD
XRATES.... Date..... us dollar
18551      15 OCT 2018    0.6506
18770      22 MAY 2019    0.6505
18796      17 JUN 2019    0.6505
18845      05 AUG 2019    0.6505
18965      03 DEC 2019    0.6500
19843      29 APR 2022    0.6504
19871      27 MAY 2022    0.6505
20116      27 JAN 2023    0.6506
20119      30 JAN 2023    0.6500
```

9 record(s) listed

We could also have used the `BETWEEN` operator to achieve the same selection:

```
SORT XRATES WITH USD BETWEEN "0.65" "0.651" DATE USD
```

The `BETWEEN` operator returns true if the field is greater than or equal to the first value specified, and less than or equal to the second value specified.

Comparison against a database value

The selection criteria used so far have compared a value in the database with a value we specify in the selection clause. There is another type of selection where we want to compare a database value against another database value.

To demonstrate this, we'll use the interest rates file that you imported earlier. Now, we normally expect longer term interest rates to be higher than short term interest rates, but this isn't always the case. We can use the query language and the selection criteria to find those dates where short term rates (30 days) are higher than longer term (90 days) rates.

```

SORT IRATES WITH DAYS30 GT DAYS90 DATE DAYS30 DAYS90
IRATES.... Date..... 30 Day Bank Bill 90 Day Bank Bill
18715      28 MAR 2019                1.86                1.83
18716      29 MAR 2019                1.86                1.85
18719      01 APR 2019                1.86                1.84
18720      02 APR 2019                1.86                1.85
18721      03 APR 2019                1.87                1.81
...

```

Direct identification of items

There is another way to select items if you know their item ID. This corresponds to the {record.id ...} element shown in the general syntax of a *SDQuery* statement shown in section Anatomy of a *SDQuery* Statement.

Using the first few item-ids from the above statement, we could write:

```

SORT IRATES '18715''18716''18719' DATE DAYS90
IRATES.... Date..... 90 Day Bank Bill
18715      28 MAR 2019                1.83
18716      29 MAR 2019                1.85
18719      01 APR 2019                1.84

3 record(s) listed

```

While this seems fairly awkward, there are situations where you know the item-ids that you want, and this becomes an easy method to extract the desired records.

Note (once again) that while the item-ids above are shown to be single-quoted, *SD* does not require this – but other multi-value databases do.

Note the following points about selection using a list of item-id’s:

- long lists of item-id’s will be cumbersome
- the item-id will need to comprise “meaningful” data if you are to have any chance of knowing it.

Sort clause

So far, we have been using the *SORT* verb, but only sorting the records into the default order. There are many cases where we need to sort the data into some other order. For example, we want to know when we had the lowest exchange rate against the US dollar:

```

SORT XRATES BY USD DATE USD
XRATES.... Date..... us dollar
20014      17 OCT 2022    0.5558
20833      13 JAN 2025    0.5566
20008      11 OCT 2022    0.5580
20009      12 OCT 2022    0.5583
20830      10 JAN 2025    0.5593
20840      20 JAN 2025    0.5595
20834      14 JAN 2025    0.5598

```

Of course, this dataset is limited to dates later than the start of 2018. If you load some of the historic data series available from the RBNZ website, you will find that the NZ Dollar reached a low of 0.3922 against the US Dollar in November 2000.

This listing shows the data sorted into ascending order. What about descending order? This uses a `BY.DSND` (note the period in the middle of the word) modifier instead of `BY`. `SD` will also accept `BY.DSND` for compatibility with other multi-value environments.

```

SORT XRATES BY.DSND USD DATE USD
XRATES.... Date..... us dollar
19415      25 FEB 2021    0.7435
18313      19 FEB 2018    0.7397
18310      16 FEB 2018    0.7393
18296      02 FEB 2018    0.7388
18309      15 FEB 2018    0.7376

```

Multiple sort values can be specified, and ascending and descending sorts freely mixed. Sorting will occur in the order specified within the statement. Consider:

```

SORT XRATES WITH USD LT "0.7349" BY.DSND USD BY DATE DATE USD
XRATES.... Date..... us dollar
18292      29 JAN 2018    0.7347
18294      31 JAN 2018    0.7347
19414      24 FEB 2021    0.7343
18315      21 FEB 2018    0.7341
18371      18 APR 2018    0.7335
18286      23 JAN 2018    0.7334
18336      14 MAR 2018    0.7334

```

This sorts the selected data into descending exchange rate order and then applies a secondary ascending sort to the date. This means that where there are multiple entries for a given exchange rate, these will be sorted into ascending date order. So, the two entries for 0.7347 are shown with the Jan 29 entry first, followed by Jan 31.

Try changing the `BY.DSND` sort into an ascending sort to make sure that the data output is sorted correctly.

Display clause

The display clause tells `SDQuery` what to display and consists of a list of dictionary items along with optional processing and formatting codes. We have used simple display clauses throughout this section to see the results of the selection and sort clauses.

To see the dictionary items that are available for any given file, simply type:

```

SORT DICT filename

```

SORT DICT XRATES							
@ID.....	TYPE	LOC.....	CONV..	NAME.....	FORMAT	S/M	ASSOC...
DATE	D	0	D	Date	12R	S	
@ID	D	0		XRATES	10L	S	
YEARX	D	0	DY	Year	4R	S	
USD	D	1	MR44,	US Dollar	7R	S	
GBP	D	2	MR44,	UK Pound	7R	S	
AUD	D	3	MR44,	Aus Dollar	7R	S	
JPY	D	4	MR22,	Jap Yen	7R	S	
EUR	D	5	MR44,	Euro	7R	S	
CAD	D	6	MR44,	Canada Dolla r	7R	S	
KRW	D	7	MR22,	SKorea Won	7R	S	
CNY	D	8	MR44,	Chinese Yuan	7R	S	
MYR	D	9	MR44,	Malay Ringgi t	7R	S	
HKD	D	10	MR44,	HK Dollar	7R	S	
IDR	D	11	MR22,	Indonesia Ru piah	9R	S	
THB	D	12	MR44,	Thai Baht	8R	S	
SGD	D	13	MR44,	Singapore Do llar	7R	S	
TWD	D	14	MR44,	Taiwan Dolla r	7R	S	
DOM	I	OCONV(@ID, 'DD')		Day	3R	S	
MTHNO	I	OCONV(@ID, 'DM')		Month	2R	S	
MONTH	I	OCONV(@ID, 'DMA')	MCT	Month	10L	S	
MTH	I	OCONV(@ID, 'DMA[3]')	MCT	Mth	3L	S	
DOW	I	OCONV(@ID, 'DW')		Day	3R	S	
DAY	I	OCONV(@ID, 'DWA[3]')	MCT	Day	3L	S	
YEAR	I	OCONV(@ID, 'DY')		Year	4R	S	

24 record(s) listed

At its simplest, a display clause is simply a list of dictionary items for the file:

SORT XRATES	DATE	USD	GBP	AUD
XRATES....	Date.....	us dollar	uk pound	aus dollar
18266	03 JAN 2018	0.7095	0.5219	0.9066
18267	04 JAN 2018	0.7086	0.5245	0.9057
18268	05 JAN 2018	0.7154	0.5275	0.9106
18271	08 JAN 2018	0.7171	0.5283	0.9119

In this example, everything after the word XRATES makes up the display clause.

Modifiers

Does the output of the ID annoy you? We can suppress the display of the item ID by using the keyword: ID.SUP OR ID-SUPP

You will note that SD often has two very similar words which do the same thing. The native dialect used by SD uses periods in these words, but words with hyphens are supported for compatibility with other multi-value environments.

SORT XRATES	DATE	USD	GBP	AUD	ID.SUP
Date.....	us dollar	uk pound	aus dollar		
03 JAN 2018	0.7095	0.5219	0.9066		
04 JAN 2018	0.7086	0.5245	0.9057		
05 JAN 2018	0.7154	0.5275	0.9106		
08 JAN 2018	0.7171	0.5283	0.9119		

That looks a better presentation.

SD has a range of keywords that modify the way the output is displayed. Some of these are listed below:

Keyword	Synonyms	Purpose
COL.HDR.SUPP	COL.HDR.SUP COL-HDR-SUPP COL-HDR-SUP	Suppresses page and column headings
COL.SUP		Suppresses column headings
COUNT.SUP		Suppresses the count of records selected
DBL.SPC	DBL-SPC	Double-spaces the output
DET.SUP	DET-SUPP	Suppresses detail lines in reports so that only totals are displayed
HDR.SUP	HDR-SUPP SUPP	Suppresses the default page heading
ID.ONLY	ONLY	Ignore the display clause and only list the ID
ID.SUP	ID-SUPP	Suppresses the display of the ID
LPTR		Send output to the printer
NEW.PAGE		Forces every record to display on a new page

You could try adding these to your queries to see what happens – but sometimes, they are best used in specific circumstances. On our reports so far, `DET.SUP` will suppress all the output lines, but will be useful with more advanced reports. Using `LPTR` may not work well until we define a printer.

Grouping records

Often, we want to display information over a period of time, but be able to break that report into blocks of information. Therefore, each year, product, or customer will be separated from other years, products, or customers.

This grouping is often (but not always) accompanied by a summary of information relating to that group – such as the total of sales for a given year or customer. This section will cover how to group information, while the next section will cover how to summarise the information.

Grouping data involves two steps – sorting the data so that related items appear together, and then placing a break in the output to visually group the items. Sorting has already been covered, so the new concept is breaking the data into groups.

The keyword that *SD* uses to break the data into groups is `BREAK.ON` (or `BREAK-ON`), and has the following general format:

```
BREAK.ON { "options" } dict-item
```

A simple statement breaking the exchange rate data into groups would be:

```

SORT XRATES BREAK.ON YEAR BREAK.ON MONTH DATE USD ID.SUP
Year Month..... Date..... us dollar
2018 January 03 JAN 2018 0.7095
2018 January 04 JAN 2018 0.7086
2018 January 05 JAN 2018 0.7154
etc
2018 January 25 JAN 2018 0.7351
2018 January 26 JAN 2018 0.7321
2018 January 29 JAN 2018 0.7347
2018 January 30 JAN 2018 0.7323
2018 January 31 JAN 2018 0.7347
**
2018

2018 February 01 FEB 2018 0.7368
2018 February 02 FEB 2018 0.7388
2018 February 05 FEB 2018 0.7301
2018 February 07 FEB 2018 0.7333
2018 February 08 FEB 2018 0.7212

```

Most of the January entries have been removed to reduce the size of the listing.

At the end of each group, two asterisks are placed in the column of the field causing the break. The command said to break on year and month. We can see the asterisks in the month column, and if we followed the listing to the point where the year changes, we would see the asterisks appear there too.

```

SORT XRATES BREAK.ON YEAR BREAK.ON MONTH DATE USD ID.SUP
Year Month..... Date..... US Dollar
etc
2018 December 27 DEC 2018 0.6732
2018 December 28 DEC 2018 0.6711
2018 December 31 DEC 2018 0.6713
**
2018
**

2019 January 03 JAN 2019 0.6620
2019 January 04 JAN 2019 0.6682
2019 January 07 JAN 2019 0.6733

```

What say we don't want the asterisks to appear. There are a couple of ways we can get rid of them. Consider:

```

BREAK.ON MONTH " "
BREAK.ON MONTH "'L'"

```

The first method shown above simply replaces the asterisks with the text between the quotation marks. In this case, that text is a single space, so nothing is displayed.

The second method suppresses the break line. This has the effect of moving the groups of data closer together:

```

SORT XRATES BREAK.ON "'L'" YEAR BREAK.ON "'L'" MONTH DATE USD ID.SUP
Year Month..... Date..... US Dollar
etc
2018 January 26 JAN 2018 0.7321
2018 January 29 JAN 2018 0.7347
2018 January 30 JAN 2018 0.7323
2018 January 31 JAN 2018 0.7347

2018 February 01 FEB 2018 0.7368
2018 February 02 FEB 2018 0.7388
2018 February 05 FEB 2018 0.7301
2018 February 07 FEB 2018 0.7333

```

You can also use the 'O' option to only show the break value when it first occurs. This is another way of visually separating the groups:

```

SORT XRATES BREAK.ON "'L'" YEAR BREAK.ON "'LO'" MONTH DOM USD ID.SUP
Year Month..... Day us dollar
2018 January 03 0.7095
2018 04 0.7086
2018 05 0.7154
etc
2018 25 0.7351
2018 26 0.732
2018 29 0.7347
2018 30 0.7323
2018 31 0.7347

2018 February 01 0.7368
2018 02 0.7388
2018 05 0.7301
2018 07 0.7333

```

We didn't specify an 'O' option for the year break because too much data would pass between the each break, and we would be left wondering which year we were looking at.

We could further add a 'P' option to the break. This would cause a page break to occur whenever the break-point was reached. In other words, once all the January data was displayed, it would pause until you pressed enter before displaying February on a fresh screen.

The introduction to this section noted that the first thing we had to was sort the data. But none of these commands actually specifies a sort criteria – although they use the SORT verb. The reason that no sort criteria is specified is that sorting by the ID (which is what the SORT verb does after all other sort criteria have been implemented) puts the records in ascending date order – which is what we want. Therefore, no explicit sort order needs to be specified.

The BREAK.ON clause contains a number of other useful options, but we need to explore these in combination with the creation of summary data. So, we'll look at that now.

Generating summary information

Typical summary information that is generated from grouped data includes totals, averages, and percentages. SD provides keywords to calculate these summaries as well as a few others. The table below shows the available summary keywords, and their synonyms:

Keyword	Synonyms	Purpose
AVERAGE	AVG	Averages the specified field
CUMULATIVE		Reports the cumulative value of the field
ENUMERATE	ENUM	Counts the values in the field
MAX		Reports the maximum value of the field in the group
MIN		Reports the minimum value of the field in the group
PERCENTAGE	PERCENT PCT %	Reports the field value as a percentage of the field total
TOTAL		Reports the total of the field

The general format used by these keywords is as follows:

```
keyword dict-item { field qualifiers }
```

The field qualifiers have a range of functions. Some of these modify the way the keywords operate (e.g. NO.NULLS tells the AVERAGE keyword to ignore null items), while others specify how to display the results (e.g. CONV, FMT, COL.HDG).

MIN, MAX, AVG

A simple set of summary data from the exchange rate file would be:

```

SORT XRATES BREAK.ON "'UV'" YEAR BREAK.ON "'UV'" MONTH DOM MIN USD AVG USD MAX USD ID.SUP
Year Month..... Day us dollar us dollar us dollar
2018 January 03 0.7095 0.7095 0.7095
2018 January 04 0.7086 0.7086 0.7086
etc
2018 January 30 0.7323 0.7323 0.7323
2018 January 31 0.7347 0.7347 0.7347
2018 January 0.7086 0.7255 0.7356

2018 February 01 0.7368 0.7368 0.7368
2018 February 02 0.7388 0.7388 0.7388
2018 February 05 0.7301 0.7301 0.7301

```

This listing tells us that the average exchange rate against the US dollar in January of 2018 was approximately 72.6 US cents to the NZ dollar, with a minimum value during the month of about 70.9 US cents and a maximum of about 73.6 US cents.

Note that the BREAK-ON clause has options of 'UV'. The 'v' tells SD to display the value of the break field in the break line, while the 'U' means display a row of underline characters between the group and the break line.

Note also that these option codes are contained within single quote marks inside a pair of double quote marks. This is explained by the general format of these BREAK.ON options:

```
"text 'codes'"
```

The text is used to replace the asterisks on the break line (covered in the previous section), while the codes provide other instructions to SD. The single quote marks are necessary to distinguish the codes from the text. If the text itself contains a single quote mark, then this should be entered as two single quote marks.

Suppressing detail lines

So far, all queries have used displayed all of the selected records. However, we frequently want to display only summary information without showing the individual record detail. Once again, SD has another keyword to enable this:

```
DET.SUP or DET-SUPP
```

For example, to simply show the monthly summary data for the same query as shown above, we would use:

```

SORT XRATES WITH YEAR EQ "2018" BREAK.ON MONTH DOM MIN USD AVG USD MAX USD DET.SUP
Month..... Day us dollar us dollar us dollar
January      0.7086      0.7255      0.7356
February     0.7212      0.7312      0.7397
March        0.7184      0.7257      0.7334
April        0.7064      0.7258      0.7373
May          0.6863      0.6953      0.7051
June         0.6741      0.6941      0.7042
July         0.6703      0.6788      0.6851
August       0.6564      0.6671      0.6805
September    0.6511      0.6595      0.6683
October      0.6427      0.6530      0.6621
November     0.6540      0.6766      0.6866
December     0.6711      0.6829      0.6943
=====
              0.6427      0.6921      0.7397

```

251 record(s) listed

Note that we've dropped the ID.SUP from the above statement, because it is implied by the use of DET.SUP. You can leave it there if you want – it makes no difference.

There is another curiosity with the above listing. The DOM field has generated an entry on the break line even though it does not do this in the detail listing. This is actually the last 'day of month' value before the break.

Formatting column headings

Now, while this shows the summarised data as we want, it is pretty confusing to have each column headed "US Dollar". We really need to change the headings of the columns to make it clear which column represents what data. This is where we use some of the field qualifiers referred to above:

```

SORT XRATES WITH YEAR EQ "2018" BREAK.ON MONTH DOM MIN USD COL.HDG "Min USD" AVG USD COL.HDG "Avg
USD" MAX U
Month..... Day Min USD Avg USD Max USD
January      0.7086      0.7255      0.7356
February     0.7212      0.7312      0.7397
March        0.7184      0.7257      0.7334
April        0.7064      0.7258      0.7373
May          0.6863      0.6953      0.7051
June         0.6741      0.6941      0.7042
July         0.6703      0.6788      0.6851
August       0.6564      0.6671      0.6805
September    0.6511      0.6595      0.6683
October      0.6427      0.6530      0.6621
November     0.6540      0.6766      0.6866
December     0.6711      0.6829      0.6943
=====
              0.6427      0.6921      0.7397

```

251 record(s) listed

This uses the COL.HDG keyword to supply a more appropriate column heading for the maximum, minimum and average exchange rates. The format of the COL.HDG keyword is as follows:

field COL.HDG "text"

where field is the dictionary item¹³ for which we wish to apply the new column heading, and text is the new column heading. The text must be enclosed in either single or double quotes. The text may also contain some formatting options.

¹³ COL.HDG can also apply to a calculated field that is specified entirely in the SDQuery statement rather than in a dictionary item. Creation of this type of field is covered later.

While *SD* allows a choice of either single or double quotes, a standard convention would be to use double quotes (as shown above), and to use single quotes to delimit options within the text.

Three formatting codes are allowed for COL.HDG. These are:

Code	Purpose
'L'	Breaks the text into multiple lines
'R'	Right-aligns the column heading
'X'	Suppresses the dot fillers normally inserted into column headings

To illustrate the use of the these options, consider:

```

SORT XRATES WITH YEAR EQ "2018" BREAK.ON MONTH MIN USD COL.HDG "'R'Min'L'USD" AVG USD COL.HDG
"'R'Avg'L'USD"
Month.....      Min      ...Avg      Max....
                USD      ...USD      USD....
January         0.7086    0.7255    0.7356
February        0.7212    0.7312    0.7397
March           0.7184    0.7257    0.7334
April           0.7064    0.7258    0.7373
May             0.6863    0.6953    0.7051
June            0.6741    0.6941    0.7042
July            0.6703    0.6788    0.6851
August          0.6564    0.6671    0.6805
September       0.6511    0.6595    0.6683
October         0.6427    0.6530    0.6621
November        0.6540    0.6766    0.6866
December        0.6711    0.6829    0.6943
=====
                0.6427    0.6921    0.7397

251 record(s) listed

```

This uses the 'L' option to split the column headings into two lines. It then uses the 'R' and 'X' options on the first column to right-justify the heading and suppress the dot fillers. The second column only uses 'R' to right-justify, while the final column doesn't use either of these options.

This demonstrates that column headings can be broken into multiple lines. Of course, it would be inconvenient if we always had to use a COL.HDG keyword to do this – there must be a way to do this directly in the dictionary. And of course, there is:

Consider:

```

SORT IRATES WITH YEAR EQ "2018" BREAK.ON MONTH MIN OVERNIGHT AVG OVERNIGHT MAX OVERNIGHT ID.SUP
DET.SUP
YEAR is not a field name or expression

```

It seems that we haven't defined YEAR in the IRATES dictionary yet. And of course, there were a number of other dictionary items we defined in XRATES that are not in IRATES. Now, because these dictionaries work off the ID of each item, and because the ID's used in the IRATES file are identical to those used in the XRATES file, we can just copy these items from the XRATES dictionary to the IRATES dictionary.

```

COPY FROM DICT XRATES TO DICT IRATES YEAR MTHNO MONTH MTH DOM DOW DAY
7 record(s) copied.

```

The COPY command has a number of formats and options. See the documentation or online help for details.

Now, let's try again:

```

SORT IRATES WITH YEAR EQ "2018" BREAK.ON MONTH MIN OVERNIGHT AVG OVERNIGHT MAX OVERNIGHT ID.SUP
DET.SUP                               Page 1
Month..... Overnight Cash Rate   Overnight Cash Rate   Overnight Cash Rate
January                1.50                1.66                1.76
February              1.50                1.67                1.76
March                  1.50                1.67                1.80
April                  0.00                1.62                1.79
May                    1.64                1.73                1.85
June                   1.59                1.69                1.78
July                   1.57                1.67                1.77
August                 1.59                1.71                1.82
September              1.50                1.68                1.76
October                0.00                1.60                1.82
November               1.61                1.72                1.79
December               1.64                1.72                1.76
=====
                        0.00                1.68                1.85

```

251 record(s) listed

Clearly, the column heading is much wider than the actual data. It would be much better if the heading were split between the words “Overnight” and “Cash”.

Go back to the MODIFY editor, and display the OVERNIGHT dictionary item. We want to modify line 4 (the NAME), so type in 4, and the heading will be displayed at the bottom of the screen. Use the arrow keys to move the cursor to the space character, and delete it using the Delete key. Now, hold down the Ctrl key and press Q. Release the Ctrl key. You will note the SD is prompting you with “Quote char” at the bottom of the screen. Press V. You will now see another character appear where the space character used to be. The actual character you see will depend on the character set in use by your computer, but a y with two dots over the top is common (ÿ). Press enter, and your modified column heading will appear in the main dictionary display. Type in F1 to file the item, and enter to return to the colon prompt.

What have we done? We have inserted a Value Mark into the heading. This should act to split the column heading into two lines.

```

SORT IRATES WITH YEAR EQ "2018" BREAK.ON MONTH MIN OVERNIGHT AVG OVERNIGHT MAX OVERNIGHT ID.SUP
DET.SUP
Month..... Overnight  Overnight  Overnight
              Cash Rate  Cash Rate  Cash Rate
January                1.50                1.66                1.76
February              1.50                1.67                1.76
March                  1.50                1.67                1.80
April                  0.00                1.62                1.79
May                    1.64                1.73                1.85
June                   1.59                1.69                1.78
July                   1.57                1.67                1.77
August                 1.59                1.71                1.82
September              1.50                1.68                1.76
October                0.00                1.60                1.82
November               1.61                1.72                1.79
December               1.64                1.72                1.76
=====
                        0.00                1.68                1.85

```

251 record(s) listed

:

Value marks are one of a number of system delimiters used by SD and multi-value databases in general. Indeed the “value” in “Value Mark” is the origin of the term “multi-value”. Value marks delimit the boundaries of each value of information in items (fields) so that we can store, retrieve, and manipulate multi-valued items.

Clearly, most of the other items in the IRATES dictionary can have their column headings split into two lines. Note that you can put more than one value mark in the heading to make 3 or 4 line headings, or even more if you so wish.

Now – there is something wrong with the above listing. The minimum column is showing an average of 0.00. This is due to null values found in IRATES. The default behavior is to treat null items as zero. If we list the data for the overnight cash rate, we find that quite a number of days have no quoted value. This is a situation where we should use the NO.NULLS modifier:

```

SORT IRATES WITH YEAR EQ "2018" BREAK.ON MONTH MIN OVERNIGHT NO.NULLS AVG OVERNIGHT NO.NULLS MAX
OVERNIGHT ID.SUP DET.SUP
Month..... Overnight Overnight Overnight
           Cash Rate Cash Rate Cash Rate
January           1.50      1.66      1.76
February          1.50      1.67      1.76
March             1.50      1.67      1.80
April             1.55      1.71      1.79
May               1.64      1.73      1.85
June              1.59      1.69      1.78
July              1.57      1.67      1.77
August            1.59      1.71      1.82
September         1.50      1.68      1.76
October           1.50      1.68      1.82
November          1.61      1.72      1.79
December          1.64      1.72      1.76
           =====
           1.50      1.69      1.85

251 record(s) listed

```

We have now applied the NO.NULLS keyword to both the MIN and the AVG columns, but haven't bothered with the MAX column. This is because the presence of a null has no effect on what value is returned for a maximum. However, you can compare the two listings to see the other two columns clearly have been affected.

You may think that perhaps we should always use NO.NULLS – but it really depends on the data that you have, and what you are trying to measure. If you are trying to get the average income of a group, and some of that group have no income, then their zero or null income should be included as part of the average (or minimum) value.

Totaling data

Another keyword that is important in reporting summary information is TOTAL. We haven't used that yet because interest and exchange rate data isn't good for totaling. However, we also loaded a file named TEX.QCH that contains better data for totaling. Most of the dictionary items to do this have already been created. You can view these by typing:

```
SORT DICT TEX.QCH
```

Let's look at some of those dictionary items. The CTRY dictionary item looks like:

```

I
FIELD(@ID, '*', 2)

Ctry
2L
S

```

Before we proceed further, it is useful to know that the @ID field of TEX.QCH has a structure of:

```
YYYYQ * CC * HH
```

where YYYYQ is the year and quarter; CC is a country identifier; and HH is an HS code. There are no spaces between the elements.

The FIELD function extracts a field from a delimited string. In this case, we have specified that the source string is the @ID field, the delimiter is an asterisk (*), and we want the second field.

This expression could also have been written as:

```
OCONV(@ID, 'G1*1')
@ID[7,2]
@ID['*', 2, 1]
```

The first of these is termed a “Group conversion” which is analogous to the FIELD function, but uses a different syntax. The ‘G’ indicates a group conversion; the first ‘1’ means skip one field; the asterisk is the delimiter character; and the final ‘1’ means return one field.

The second alternative is simply a string extraction. It says return two characters from the @ID, starting at position 7. This relies on the components of the @ID being fixed in length, whereas the other versions extract the second field delimited by asterisk characters.

The third expression¹⁴ looks like a string extraction because it uses the square brackets, but it is actually a group extraction. This statement says extract 1 group from @ID, delimited by the asterisk character, starting at group 2.

Which expression you use is up to you. But you need to be aware that there is more than one way to achieve a given end, and that other people may use a different expression than you would.

The YYYYQ and HS dictionary items are very similar to CTRY – except that they extract a different part of the @ID. The YEAR and QTR dictionaries take the field returned by YYYYQ and carry out substring extractions to get the year and quarter numbers.

We’ll deal with the other dictionary items later.

Now, let’s look at total exports for a year, broken into country destinations:

```
SORT TEX.QCH WITH YEAR EQ "2012" BY CTRY BREAK.ON CTRY TOTAL FOB DET.SUP
Ctry      .....FOB Value
AD                0
AE          579,039,997
AF           78,642
etc
ZA          244,036,303
ZM           88,493
ZR                0
ZW          1,763,241

          44,356,210,157

96824 record(s) listed
```

14 This syntax is not supported in the GPL version of SD.

This query statement used the DET.SUP keyword to suppress the individual record detail, so that only the country totals are displayed. If you look at the count of records in the report, you will realise that a lot of records have been summarised in this report.

If we take the DET.SUP keyword out of the statement, then we find that many records actually contain no data:

```

SORT TEX.QCH WITH YEAR EQ "2012" BY CTRY BREAK.ON CTRY TOTAL FOB
TEX.QCH..... Ctry .....FOB Value
20121*AD*01   AD
20121*AD*02   AD
20121*AD*03   AD

```

We could exclude these records with:

```

SORT TEX.QCH WITH YEAR EQ "2012" AND WITH FOB GT "0" BY CTRY BREAK.ON CTRY TOTAL FOB DET.SUP
Ctry .....FOB Value
AE          579,039,997
AF           78,642
AG          726,777
etc
ZA          244,036,303
ZM           88,493
ZW          1,763,241

          44,356,210,157

17023 record(s) listed

```

There are two things to note here. Firstly, the record count has shrunk from over 968,000 down to just over 17,000 – so we’ve excluded a lot of empty records. The second thing to note is that because we’ve excluded those records, not all countries appear in the listing. For example, country ‘AD’ (Andorra) is shown in the first listing but not the second. This means that New Zealand didn’t export anything to Andorra during 2012.

This raises an important issue regarding database design. If we want to display information about an entity (country), then we need to record that data *even when that data is empty*. Or *the absence of data can be just as important as its presence*.

If we only recorded actual exports in this database, we would not be able to produce a listing of exports that included ALL countries. Someone who didn’t have knowledge of our database wouldn’t be sure whether we didn’t export to that country in the period, or whether we simply hadn’t recorded the data. So, there is value to data, even when the data is null.

In this database design, we store a lot of empty records. On the other hand, the user now has the choice of displaying all countries in a report, or only those countries for which there is genuine data. We can even answer the question “Which countries did New Zealand NOT export to in 2012?”

Change the entity to make the example more relevant to you. For example, in a hotel booking system, we want to record the status of all rooms in the hotel – not just those rooms that have a booking.

Let’s show a bit more detail in this report:

```

SORT TEX.QCH WITH CTRY EQ "AU""GB""JP""DE""US" AND WITH YEAR EQ "2012" BY CTRY BY HS BREAK.ON CTRY
BREAK.ON HS TOTAL FOB DET.SUP
Ctry  HS Code  . . . . .FOB Value
      01      78,985,296
      02      65,252,009
      03      230,202,508
etc
      98      104,033,980
      99              0
AU      -----
      9,159,794,657
      01              0
      02      343,692,327
etc

```

The report now shows exports for each country broken into major categories according to the Harmonised System (HS) for export classification. The above listing shows that country 'AU' (Australia) took \$9,159m of exports during 2012, of which \$230m was fish (HS code 03).

Page breaks

In the above listing, you may want greater separation for each country. We can do this by inserting a page break at each country break by using the 'P' option in the break command:

```

SORT TEX.QCH WITH CTRY EQ "AU""GB""JP""DE""US" AND WITH YEAR EQ "2012" BY CTRY BY HS BREAK.ON
""P"" CTRY BREAK.ON HS TOTAL FOB ID.SUP DET.SUP
Ctry  HS Code  . . . . .FOB Value
      01      78,985,296
      02      65,252,009
      03      230,202,508
etc
      98      104,033,980
      99              0
AU      -----
      9,159,794,657

```

The next page would show the data for Germany (country 'DE'), while the one after that will show the UK data (country 'GB'), and so on.

One problem with these listings is that the country information is not being displayed until the break line. That is several screens of information before you find which country you are displaying.

One solution to this is to include the country identifier with the HS chapter identifier to create a compound attribute. We'll use an I-type dictionary named CTRY.HS to do this with an expression of:

```
CTRY:' ':HS
```

This concatenates the CTRY and HS dictionaries, with a space character between them. The format code becomes '5L' to accommodate the width of the combined item. Now:

```

SORT TEX.QCH WITH CTRY EQ "AU""GB""JP""DE""US" AND WITH YEAR EQ "2012" BY CTRY BY HS BREAK.ON
""P"" CTRY BREAK.ON CTRY.HS TOTAL FOB DET.SUP
Ctry  Ctry HS  . . . . .FOB Value
AU 01      78,985,296
AU 02      65,252,009
AU 03      230,202,508
etc
AU 98      104,033,980
AU 99              0
AU      -----
      9,159,794,657

```

Now, we may decide that we don't want the main country column displaying, but we still want to break on that value. We use the BREAK.SUP keyword for this:

```

SORT TEX.QCH WITH CTRY EQ "AU""GB""JP""DE""US" AND WITH YEAR EQ "2012" BY CTRY BY HS BREAK.SUP
""P"" CTRY BREAK.ON CTRY.HS TOTAL FOB DET.SUP
Ctry HS      .....FOB Value
AU 01        78,985,296
AU 02        65,252,009
AU 03        230,202,508
etc
AU 98        104,033,980
AU 99         0
-----
              9,159,794,657

```

Grand totals

SD automatically creates a grand total line for your query when you use the TOTAL, AVG, ENUM, MAX, MIN, or PCT keywords. This consists of a row of double underlines followed by the summary value.

```

SORT TEX.QCH WITH YEAR EQ "2009""2010""2011""2012" BREAK.ON ""V"" YEAR TOTAL FOB ID.SUP DET.SUP
Year      .....FOB Value
2009     37,776,649,244
2010     41,769,862,081
2011     45,905,449,806
2012     44,356,210,157
-----
          169,808,171,288
387296 record(s) listed

```

You can change the format of this grand total line by using the GRAND.TOTAL keyword. Options include providing specific text for the grand total line, suppressing the grand total line completely, or printing it on a separate page.

```

SORT TEX.QCH WITH YEAR EQ "2012" AND WITH HS EQ "02" AND WITH CTRY EQ "AU""CN""DE""GB""JP""US" BY
CTRY BREAK.ON CTRY TOTAL FOB GRAND.TOTAL "Meat" DET.SUP
Ctry      .....FOB Value
AU        65,252,009
CN        411,719,261
DE        343,692,327
GB        589,510,001
JP        280,448,620
US        1,180,039,662
Meat     =====
          2,870,661,880
24 record(s) listed

```

In the above command, the text “Meat” is inserted into the grand total line by the GRAND.TOTAL keyword. To suppress the grand total, use the ‘L’ option:

```

SORT TEX.QCH WITH YEAR EQ "2012" AND WITH HS EQ "02" AND WITH CTRY EQ "AU""CN""DE""GB""JP""US" BY
CTRY BREAK.ON CTRY TOTAL FOB GRAND.TOTAL ""L"" DET.SUP
Ctry      .....FOB Value
AU        65,252,009
CN        411,719,261
DE        343,692,327
GB        589,510,001
JP        280,448,620
US        1,180,039,662
24 record(s) listed

```

Alternatively, you can use the NO.GRAND.TOTAL modifier:

```

SORT TEX.QCH WITH YEAR EQ "2012" AND WITH HS EQ "02" AND WITH CTRY EQ "AU""CN""DE""GB""JP""US" BY
CTRY BREAK.ON CTRY TOTAL FOB NO.GRAND.TOTAL DET.SUP

```

Scaling data

All our queries have so far reported FOB exports down to the last dollar. Usually, we don't want or need that level of detail. Typically, exports will be expressed in millions of dollars (and in larger countries, perhaps billions of dollars). So, how do we do that?

We've already seen that conversions can be used to change the way that data is displayed. And that is exactly what we want to do here. We want to change the way that the data is displayed without actually changing it internally.

Our FOB dictionary item currently uses a conversion of MR,. This could also be written as MR0, or MR00,. Essentially, this means to display the data with zero decimal places, and no scaling. What we want is to display the data in millions of dollars, and to display this to one decimal place.

We can do this with a conversion code of MR16,. This means that we want to descale the value by six decimal places, and display to one decimal place. We can put this in a CONV modifier to test it:

```

SORT TEX.QCH WITH HS EQ "02" AND WITH CTRY EQ "AU""CN""DE""GB""JP""US" AND WITH YEAR EQ "2012" BY
CTRY BREAK.ON CTRY TOTAL FOB CONV "MR16Z," COL.HDG "'RX'FOB $m" DET.SUP
Ctry          FOB $m
AU             65.3
CN            411.7
DE            343.7
GB            589.5
JP            280.4
US            1180.0
=====
                2870.7

24 record(s) listed
```

This has also used a COL.HDG modifier to show that the values are now being expressed in millions of dollars.

If you compare the above listing with those found on previous pages, you will find that both the individual values and the grand total have been correctly expressed in millions rounded to one decimal place.

As this is a common output format, we should put this into a dictionary item. Let's call it FOB.M:

```

I
FOB
MR16,
'R'FOB $m
9R
S

SORT TEX.QCH WITH HS EQ "02" AND WITH CTRY EQ "AU""CN""DE""GB""JP""US" AND WITH YEAR EQ "2012" BY
CTRY BREAK.ON CTRY TOTAL FOB.M DET.SUP
AU             65.3
CN            411.7
DE            343.7
GB            589.5
JP            280.4
US            1,180.0
=====
                2,870.7

24 record(s) listed
```

Note that this dictionary item has also been given an output width that is more appropriate to the size of data that is appearing.

Page headings and footings

From the *SDQuery* elements introduced so far, you should be able to construct a basic report that selects, sorts, groups, and summarises data. However, to make this look like a proper report, we need to add page headings and footings. The keywords to do this in *SD* are `HEADER` and `FOOTING`, and have the following format:

```
HEADING "text"  
FOOTING "text"
```

where "text" contains the text to be displayed in the heading or footing, along with a number of optional formatting codes. These formatting codes are enclosed in single quotes within the text string.

Some of the formatting codes are listed below:

Code	Purpose
B	Inserts the value of the data from the corresponding B code in a <code>BREAK.ON</code> option string
C	Centres the heading text
D	Inserts the date
F	Inserts the file name
G	Inserts a gap to utilise the full width of the output device
L	Inserts a new line
P	Inserts the page number

A full list of formatting codes can be found in the *SD* documentation and on-line help.

For example:

```
TERM 65  
SORT XRATES WITH YEAR EQ "2018" BREAK.ON MONTH AVG USD NO.NULLS AVG AUD NO.NULLS AVG GBP NO.NULLS  
DET.SUP HEADING "'DGC'Average Monthly Exchange Rates'G'Page 'PLC'Year: 2018'L'"  
TERM 132
```

produces:

```
11 FEB 2025      Average Monthly Exchange Rates      Page    1  
                  Year: 2018  
  
Month....  us dollar  aus dollar  uk pound  
January      0.7255      0.9123      0.5252  
February     0.7312      0.9277      0.5228  
March        0.7257      0.9343      0.5196  
April        0.7258      0.9432      0.5150  
May          0.6953      0.9239      0.5156  
June         0.6941      0.9265      0.5225  
July         0.6788      0.9168      0.5155  
August       0.6671      0.9100      0.5178  
September    0.6595      0.9160      0.5054  
October      0.6530      0.9186      0.5017  
November     0.6766      0.9342      0.5244  
December     0.6829      0.9503      0.5394  
=====      =====      =====  
              0.6921      0.9257      0.5185  
  
251 record(s) listed
```

The first `TERM` statement adjusts the width of the output device (screen) to something appropriate for the centring of the text. The second `TERM` statement returns the screen width to its previous value.

The `HEADING` statement specifies a 3 line heading. However, there is nothing in the third line, so this simply spaces the heading from the body of the report.

The first line has 3 components – the date (specified by the 'D' option), a main heading, and a page number (specified by the 'P' option). These are separated by 'G' options which force the heading to occupy the total width of the output device (the screen), while the second element is centred by the 'C' option.

The second line only has one element which is centred by the 'C' option.

Let's extend the above statement to place the year in the heading automatically from a BREAK.ON clause:

```

TERM 65
SORT XRATES WITH YEAR GE "2018" BREAK.ON "'B'" YEAR BREAK.ON MONTH AVG USD NO.NULLS AVG AUD
NO.NULLS AVG GBP NO.NULLS DET.SUP HEADING "'DGC'Average Monthly Exchange Rates'G'Page 'PLC'Year:
'BL'"
TERM 132

```

This produces:

```

11 FEB 2025      Average Monthly Exchange Rates      Page    1
                  Year: 2018

Year  Month.....  us dollar   aus dollar  uk pound
2018  January      0.7255     0.9123     0.5252
2018  February     0.7312     0.9277     0.5228
2018  March        0.7257     0.9343     0.5196
2018  April        0.7258     0.9432     0.5150
2018  May          0.6953     0.9239     0.5156
2018  June         0.6941     0.9265     0.5225
2018  July         0.6788     0.9168     0.5155
2018  August       0.6671     0.9100     0.5178
2018  September   0.6595     0.9160     0.5054
2018  October     0.6530     0.9186     0.5017
2018  November    0.6766     0.9342     0.5244
2018  December    0.6829     0.9503     0.5394
2018                      0.6921     0.9257     0.5185

```

251 record(s) listed

The key change in this statement is the inclusion of a 'B' option in the BREAK.SUP clause. This is matched by another 'B' option in the HEADING clause. This puts the break value into the heading. Therefore, the heading correctly identifies the year whose months are displayed in the report.

There is another case of an implied option being used here. We didn't actually specify that there should be a page break on a year change, but this option is implied by putting the break value into the heading. Once again, other multi-value databases will probably require you to specify the 'P' option in the BREAK.ON clause.

Footings are included in a similar fashion. For example the following footing could be added to the above statement:

```

FOOTING "Source: Rush Flat Consulting'L'      Based on RBNZ data"

```

This would be output as:

```

Source: Rush Flat Consulting
Based on RBNZ data

```

Note that the footing appears at the bottom of the output device (screen or printed page) rather than immediately following the body of the report.

Printing and Report Styles

Printing

So far, we have found how to create a report and display it on the screen. That is good, but we often want a printed copy of the report too. So, how do we send the report to the printer?

Well, the answer is simple – there is another keyword to send the report to the printer – but there are a number of complexities associated with printing.

The LPTR keyword

To send a report to the printer, all we need to do is add the LPTR keyword to the *SDQuery* statement. The full format of the keyword is:

```
LPTR {unit}
```

where unit is the print unit number. If unit is omitted, then print unit 0 is used.

The exact meaning of print units will be covered shortly. But first, let's look at a *SDQuery* statement that sends output to the printer:

```
SORT XRATES WITH YEAR GE "2018" BREAK.ON"B" YEAR BREAK.ON MONTH AVG USD NO.NULLS AVG AUD  
NO.NULLS AVG GBP NO.NULLS DET.SUP HEADING "'DGC'Average Monthly Exchange Rates'G'Page 'PLC'Year:  
'BL'" LPTR
```

If you execute this statement, perhaps something will come out of the printer, or perhaps not. It depends on how your print units are set, and what sort of printer you have.

Print units

SD uses logical print units to provide flexibility in report output. A logical print unit may direct output to a printer, to a file, or both. Individual print units may be defined to allow printing to different printers, and in different formats (portrait, landscape, different fonts, different page sizes).

Up to 256 print units – numbered 0 to 255 – can be defined for any session. It is unlikely that you will need to define anywhere near this number of print units.

To take advantage of all of the printing features of *SD*, a PCL printer is required. If you have a GDI printer, features such as control of page orientation and print pitch are not available – but are still available to you if you define a print unit as PCL and print through a PCL to Windows print driver (such as Anzio Print Wizard, or PageTech PCLReader). Given the availability of this software (with PCLReader being free), further discussion here will assume use of the PCL settings.

To see the print units currently defined, type:

```
SETPTR DISPLAY
```

If you haven't defined any print units, you will see the following:

```

SETPTR DISPLAY
Unit Width Depth Tmgn Bmgn Mode Options
  0    80    66    0    0    1

```

Print unit 0 is the default print unit. Failing any other definition, it is defined as a page 80 characters wide by 66 lines deep, with top and bottom margins of 0 lines. Mode 1 means that output will be sent to a printer, and with no printer being specifically defined here, it will go to the default printer. No options are specified.

Let's change this definition to something a bit more useful. Type in:

```
SETPTR 1,90,66,0,2,3, AS NEXT SDPRINT, PCL, CPI 12, LEFT.MARGIN 2
```

SD will respond:

```

SETPTR 1,90,65,0,2,3, AS NEXT SDPRINT, PCL, CPI 12, LEFT.MARGIN 2
PRINT UNIT 1
  Page width      : 90
  Lines per page  : 66
  Top margin      : 0
  Bottom margin   : 2
  Mode            : 3 (Hold file: $HOLD SDPRINT)
                  Using next suffix number
LEFT.MARGIN 2
PCL: CPI = 12, LPI = 6, Weight = MEDIUM, Symbol set = ROMAN-8
Paper size = A4

```

```
OK to set new parameters (Y/N)?
```

Before you answer 'Y', consider what paper size you use. The A4 papersize being set here is fine for European countries, but North America typically uses Letter sized paper. In that case, the command should be:

```
SETPTR 1,90,61,0,2,3, AS NEXT SDPRINT, PCL, CPI 12, LEFT.MARGIN 2, PAPER.SIZE LETTER
```

This defines the output page as 90 characters wide by 61 lines deep (letter size paper is shorter than A4 paper), with a top margin of 0, and a bottom margin of 2. The output mode is 3 – meaning that the report will not be printed, but will be written to the \$HOLD file in the account that you are working in. It will be saved there with a name of SDPRINT_nnnn where nnnn is a number that is incremented with each print job. We have said that the printer is PCL capable, and that we want to print in a 12 pitch font, and have a left margin on the page of 2 characters. The paper size is A4 unless we specify a different paper size in the SETPTR command.

Now, if we execute our *SDQuery* statement:

```

SORT XRATES WITH YEAR GE "2018" BREAK.SUP""B"" YEAR BREAK.ON MONTH AVG USD NO.NULLS AVG AUD
NO.NULLS AVG GBP NO.NULLS DET.SUP HEADING "'DGC'Average Monthly Exchange Rates'G'Page 'PLC'Year:
'BL'" FOOTING "Source: Rush Flat Consulting'L" Based on RBNZ data" LPTR 1

```

... nothing appears to happen. The cursor simply returns to the command prompt. However, if you look in the \$HOLD file, then you will see a print file in there named SDPRINT_0001 (or some other number).

```

SORT $HOLD
$HOLD.....
SDPRINT_00
01

1 record(s) listed

```

If you open this file with a text editor, you will find it contains the report output, plus some HP PCL formatting codes.

If you have a PCL viewer, then you can view the contents of the report by opening this file with the viewer, and then print the report to any printer.

Now, what happens if we want to print a report in landscape mode? We create another print unit for landscape printing:

```
SETPTR 2,132,41,0,2,3, AS NEXT SDPRINT, PCL, CPI 12, LANDSCAPE, LEFT.MARGIN 2
```

Make sure you add a PAPER.SIZE setting and adjust the line count if you are not using A4.

The differences between this and the previous definition are:

- this defines print unit 2 (rather than 1)
- the page size is 132 characters wide by 41 lines deep
- the LANDSCAPE option is specified

To use this print unit, we need to specify the print unit number in the *SDQuery* command:

```
SORT XRATES WITH YEAR GE "2018" BREAK.SUP""B"" YEAR BREAK.ON MONTH AVG USD NO.NULLS AVG AUD  
NO.NULLS AVG GBP NO.NULLS DET.SUP HEADING "'DGC'Average Monthly Exchange Rates'G'Page 'PLC'Year:  
'BL'" FOOTING "Source: Rush Flat Consulting'L' Based on RBNZ data" LPTR 2
```

Note that we haven't fully utilised the available page areas. The portrait mode report has plenty of vertical space available, while the landscape mode report has plenty of horizontal space available.

Initialising print units

Of course, you don't want to have to define a print unit every time you want to print a report. Therefore, we need a way to automatically define the print units so they are ready for your use (print units only exist for your current session – if you close your session, and then start a new session, the print units you defined above will be gone).

The best way to do this is to place your print units in the LOGIN item of the account:

```
WED VOC LOGIN
```

Add the following two lines to the LOGIN entry:

```
SETPTR 1,90,65,0,2,3, AS NEXT SDPRINT, PCL, CPI 12, LEFT.MARGIN 2, BRIEF  
SETPTR 2,132,41,0,2,3, AS NEXT SDPRINT, PCL, CPI 12, LANDSCAPE, LEFT.MARGIN 2, BRIEF
```

These are the same SETPTR commands we used above, with the addition of the BRIEF option. This suppresses the 'OK to set new parameters' prompt when defining a print unit.

Next time you log into this account, these two print units should be automatically defined for you. To check this, log out, then log back in and type:

```
SETPTR DISPLAY
```

This should report the two print units and their settings.

Defining your own print units

The above two print units will get you started in defining your own print units to suit your own purposes. Start with what printer you have attached to your system, and what paper size it uses. If it is a PCL printer, or you have PCL printing software, then define the print unit as PCL as this gives you more options.

Now, you'll need to start playing with settings to see what suits you. How many characters wide do you need your printouts? This will define whether you use portrait or landscape mode, and what printing pitch is necessary to achieve this. Note: You need PCL printing to do this.

Start with your paper size. Can your printer print in landscape mode? What print pitches can it print at? Basic HP print pitches are 10, 12, and 16.67 characters per inch, while standard lines per inch settings are 6 and 8. Even these limited settings can give you a reasonable range of print units.

Now put your fully defined print unit into the account LOGIN item so that it is defined every time you log into the account.

Spooling print files

If your print units send their output to a file (mode 3), then you need a way to print these files. Using a PCL viewer to do this has already been outlined above, but you can send them direct to the printer without using a PCL viewer. However, to do this, you need a physical print unit defined – i.e. one that points to your printer.

You could duplicate the print units you already have:

```
SETPTR 10,90,65,0,2,1, PCL, CPI 12, LEFT.MARGIN 2, BRIEF
SETPTR 11,132,41,0,2,1, PCL, CPI 12, LANDSCAPE, LEFT.MARGIN 2, BRIEF
```

This defines print units 10 and 11 (10 more than the equivalent units that go to the \$HOLD file). These don't have the 'AS NEXT ...' option as this isn't relevant to a physical printer.

To print reports from the \$HOLD file to the print unit defining the physical printer, we use the SPOOL command:

```
SPOOL file record(s) LPTR n
```

So, to print our landscape report onto the landscape physical printer:

```
SPOOL $HOLD SDPRINT_0003 LPTR 11
```

Of course, this assumes that your printer is capable of receiving the data in the file. So if you are creating PCL files but you don't have a PCL printer, then you will need to use a PCL viewer.

This command allows you to specify the file name. While the default output file is \$HOLD, you can specify an alternate file in the SETPTR command (using the AS PATHNAME option) or create print files in an alternate file direct from SDBasic.

Deleting print files

If your print units are set to send their output to a file – then over time you will build up a number of items within that file. The \$HOLD file is a directory file, so you can simply delete them using a file manager. Alternatively, you could delete them using the DELETE or CLEAN.ACCOUNT verbs:

```
DELETE $HOLD item-id
DELETE $HOLD ALL
DELETE filename item-id
DELETE filename ALL
CLEAN.ACCOUNT
```

CLEAN.ACCOUNT clears not only the \$HOLD file, but the \$SAVEDLISTS and \$COMO (if it exists) files as well. See the documentation for information on the \$COMO and \$SAVEDLISTS file.

Miscellaneous Aspects of SDQuery

Default display and phrases

Type in: **SORT VOC**

Similarly: **SORT DICT XRATES**

Notice that both of these commands produce a report with one or more output fields in the display – when we didn't specify a display clause in the statement.

SD can use a default display clause for those occasions when the user does not explicitly specify a display clause. This default display clause is contained in the dictionary of the file with an item-name of @.

So, the default item for VOC looks like:

```
CT DICT VOC @
DICT VOC @
1: PH
2: DESC FMT '60T'
```

And for the dictionary listing:

```
CT DICT.DICT @
DICT.DICT @
1: PH
2: TYPE FLD CONVERSION NAME FORMAT.CODE SMV ASSOCIATION_
3: BY TYPE.CODE BY LOC.R BY NAME
```

Before we look at the contents of these items, consider the files where these items were found.

For the VOC, the @ item was found in DICT VOC. That isn't too surprising, but if you look at the user account through a file manager, you won't find a file named VOC.DIC. However, if you issue a LISTF command, you will see that the path to the VOC dictionary is @SDSYS\VOC.DIC (where @SDSYS is a token

referring to the O/S level path to the SDSYS account). This means that all accounts on your *SD* system will share the same voc dictionary (but they each have their own individual voc data file).

Now, what about the default dictionary listing. Do dictionaries have a dictionary to define words and phrases? The answer is 'Yes', and it is named DICT.DICT. Once again, this is a shared dictionary for all accounts and has a real pathname of @SDSYS\DICT.DIC. Note the two different spellings – the real filename is DICT.DIC but is referenced in all accounts as DICT.DICT. Why the difference? Internally, all dictionaries are referred to as DICT but have a real filename of *filename*.DIC, so it is simply consistent with the rest of the file naming conventions.

Let's look at what was contained in those '@' items.

Both items had 'PH' in the first line. This identifies the item to the *SDQuery* processor as a PHRASE. Put simply, a phrase is a shortcut expression for a larger set of words. When the *SDQuery* processor encounters a phrase, it substitutes the words contained on line 2 of the phrase definition for the phrase name in the *SDQuery* statement.

Therefore, line 2 contains the default display clause for those files (the voc and any dictionary).

The default display clause for the voc contains only a single output field (DESC) and a display format expression (FMT '60T').

In contrast, the default display clause for dictionaries contains seven output items and a three-level sort clause. This indicates that phrases can contain more than just the display clause of a *SDQuery* statement.

Note also that those elements are spread across lines two and three in the phrase, when the phrase is supposed to only exist on line two. However, you can spread phrases across multiple lines, as long you terminate any lines prior to the last line with a continuation character (an underscore).

Let's try creating a default display item in the XRATES dictionary:

```
PH
WITH YEAR GE "2018"
BREAK.ON YEAR BREAK.ON MONTH
AVG USD NO.NULLS AVG AUD NO.NULLS AVG GBP NO.NULLS AVG EUR NO.NULLS AVG JPY NO.NULLS_
HEADING "New Zealand Exchange Rates Since 2018'L'" DET.SUP
```

Note that we have used extension characters to spread the phrase across multiple lines.

Now try a *SDQuery* statement without no display clause:

```
SORT XRATES
```

This now generates a basic report, complete with a heading and with the ID suppressed. This makes simple reporting somewhat easier.

Now try these statements:

```
SORT XRATES HEADING "Example heading"
SORT XRATES BY.DSND @ID
SORT XRATES WITH YEAR GE "2018"
```

The first two of these work OK, but the third doesn't.

In the first case, the `HEADING` specified in the second statement was used in preference to the default heading, while the second example changed the sort criteria. However, the selection criteria in the third example did not override that in the default display phrase.

Therefore, when you set up phrases, make sure that they work properly with all the statements you are likely to use.

We can send these default reports to the print units simply by adding `LPTR` to the statement. However, `SD` has another twist for default printing.

The default display clause for printing is actually contained in the item '@LPTR'. This is also contained in the dictionary of the file being used in the `SDQuery` statement. However, if '@LPTR' does not exist, then the default display specified in '@' will be used.

This lets you define one default display for on-screen reporting, and a different (perhaps wider) default display for printing.

Saving `SDQuery` statements for later use

You now have your completed `SDQuery` statement. But it is several lines long, and you don't want to type it in every time that you want to run that particular report. What we need now is some way to save the statement, so that you can run it again later.

`SD` provides several ways to do this.

The traditional method amongst Information style databases is to create a 'sentence' or a 'paragraph' in the `voc`, and then run the sentence or paragraph by typing in its name.

A sentence contains an 's' in the first line, and the `SDQuery` command in the second line. Therefore, a sentence named 'MY.QUERY' might look like:

```
CT VOC MY.QUERY
VOC MY.QUERY
1: S
2: SORT XRATES WITH YEAR GE "2018" BREAK.SUP "'B'" YEAR BREAK.ON MONTH AVG USD NO.NULLS AVG AUD
NO.NULLS AVG GBP NO.NULLS DET.SUP HEADING "'DGC'Average Monthly Exchange Rates'G'Page 'PLC'Year:
'BL'" FOOTING "Source: Rush Flat Consulting'L' Based on RBNZ data" LPTR
```

If you already have this command sitting on your command stack, you can get `SD` to create the sentence for you. Let's assume that the command you want to load into the sentence is in stack position 5. Type in:

```
.S MY.QUERY 5 5
```

Make sure you include the dot before the 's'. This says save lines 5 to 5 as an item in the `voc` named 'MY.QUERY'. Because there is only one line to save here, the second 5 could have been omitted.

If you specify more than one line to be saved, the `SD` will save the commands as a paragraph (rather than a sentence).

Now, typing 'MY.QUERY' from the command prompt will display the report.

To see the paragraphs and sentences stored in the voc, type:

LISTS Lists sentences

LISTPA Lists paragraphs

What about running the command if it is stored in a separate file? Well, let's create a file to hold the queries in first:

```
CREATE.FILE QUERIES DIRECTORY
```

Note that this statement defines the file as a directory file. This means that you can use any text editor to edit the items directly from the operating system – although we'll still access them like any other item from inside *SD*.

Using the text editor, enter the query then make the first line of the item 'S' or 'PA' as appropriate, then file the item.

Now you can run the item by typing either of:

```
RUN QUERIES MY.QUERY  
.X QUERIES MY.QUERY
```

Which storage method should you use for your queries? That is up to you.

If you only have a small number of saved queries, then it won't matter which method you use.

However, if you have a large number of saved queries, then it is probably better to use a distinct file for this purpose. This will help keep the voc reserved for its correct purpose – that of being a repository of keywords used by *SD*.

Introduction to SDBasic

General Considerations

What is SDBasic

SDBasic is the programming language that comes bundled with *SD*. As its name implies, it is a dialect of the BASIC language. This dialect is closely related to the BASIC dialects that come with other multi-value databases.

SDBasic has its own set of enhancements that are not in other multi-value databases. These include:

- local variables
- object-oriented programming
- socket connections
- inbuilt encryption

plus enhancements to many of the standard inbuilt functions.

SDBasic is a server-side language. Some people have described SDBasic as a ‘server side scripting language’.

SD does not support any other server side language. However, *SD* does offer two interfaces to external languages – one for *Visual Basic* and one for *C*. These interfaces allow a variety of languages to communicate with *SD*. Further, applications developed using these interfaces may split the processing between the server and the client.

What is covered here?

This is not intended to be a programming tutorial – it is assumed that the reader is generally familiar with programming principles. Further, it is assumed that the reader is familiar with event-driven programming principles.

Nevertheless, a reasonable amount of basic programming information is covered, In particular, differences from other BASIC dialects will be covered.

Coding styles

Ladybridge Systems recommend the following coding style for *SD* applications:

- Write source code in lowercase for improved readability. We have partially adopted a convention that equate token names should be in uppercase.
- Use the standard command parser (!PARSER) rather than writing your own.
- Use token names from include records rather than literal values where appropriate.
- Use CRT or DISPLAY rather than PRINT unless you want the output to go to a printer.
- *SD* is reasonably case independent. Try to preserve this by looking up VOC or dictionary items as typed and, if not found, by trying again in uppercase.
- Whatever the programming purists may say, limited careful use of GOTO is fine.
- Use meaningful label names. Numeric label names are not allowed in the master source.
- The standard message handler should be used for all output text. Use message numbers in the range 10000 - 19999. We will change these to the final message number on integration where appropriate. Pick style messages from the ERRMSG file (and hence the Pick syntax variant of STOP and ABORT) may not be used.
- Programs must conform to the locking rules such that all writes and deletes, including adding a new record, are covered by a suitable lock. Programs must operate correctly with the MUSTLOCK configuration parameter set to 1.
- Programs must not rely on settings in the \$BASIC.OPTIONS record for correct compilation.
- Programs must be adequately commented that their operation can be clearly understood for maintenance purposes.

Their purpose in publishing this guide is so that any submissions of improvements can be incorporated into the *SD* codebase with a minimum of refactoring. Most of these points can be readily adopted.

These guidelines really apply if you are coding for *SD* only. However, if there is any possibility that your application will be ported to another database, then you need to consider compatibility. You may say that you have no intention of moving to another database, but that isn't the only reason to consider compatibility – your application may be so good that users of *UniVerse*, *UniData*, *D3*, *Reality*, *jBASE*, or *mvBASE* may want to use it on their systems.

What is important for compatibility? Here are some suggestions:

- Use upper case keywords – some multi-value databases (e.g. *mvBASE*) cannot handle lower case keywords
- Be consistent with your casing of variable names. For simplicity, it is best to use either all uppercase or all lowercase. The reason for this is that variable names in other databases are usually case sensitive – therefore, *TEMP*, *Temp*, and *temp* are all different variable names in most other databases, but are treated as being the same variable in *SD*.

- Think carefully before using features of SDBasic that are not standard across the multi-value flavours. For example, no other multi-value database supports local variables or object oriented programming.
- Use keywords that are consistent between the various multi-value databases. Even where different databases have the same keyword, the keyword may not always have the same syntax, or have the same functionality. For example, in *SD*, *SWAP* is a synonym for *CHANGE* (as it is in *mvBASE*). In *UniData*, *SWAP* has the same functionality, but has a different syntax. In *UniVerse*, *SWAP* has an entirely different function. In this case, do not use the keyword *SWAP* if you want to guarantee compatibility with other databases.

In this guide, the following conventions will be used:

- Keywords will be in upper case
- Variables will be in lower case
- Constants will be in upper case

You may also want to read the coding standards article on the PickWiki website:
<http://www.pickwiki.com/cgi-bin/wiki.pl?CodingStandards>

Where is the GUI?

SD does not directly support GUI programming. While this may surprise some, this is consistent with its role as a database server system. Few (if any) database servers directly support GUI programming – most rely on external toolsets to provide a GUI front-end to the database using the inbuilt API's.

There are various ways to add a GUI to *SD* applications. These include:

- Use *SDClient* to connect an external language to *SD* using either the *VB* or *C* interface

General Programming Issues

Creating, Compiling, and Running Programs

Programs are stored in an *SD* file. In the multi-value world, this file is traditionally named *BP*, but may take on any valid filename. Programs may be stored in multiple files, thereby assisting application development by keeping programs logically grouped together.

Let's create a file for programs:

```
CREATE.FILE BP DIRECTORY
Created DICT part as BP.DIC
Created DATA part as BP
Added default '@ID' record to dictionary
```

By convention, programs are often stored in a file named `BP` (meaning Basic Programs), but there is no restriction on filenames for programs. You could use `PROGRAMS`, `PROGS`, `UTILS`, or any other valid filename.

Note that we've created this file as a directory. This type of file is suitable for text items, and allows the items in the file to be edited directly from the operating system level – i.e. you can use your favourite text editor to edit the programs. I

Now, let's create the standard "Hello world" program:

```
SDEdit: BP HELLO  
Edit using <M>icro, <N>ano, <E>xit :?
```

This will start the `MICRO` editor. If the `HELLO` program exists, then `MICRO` will display the existing program; otherwise, it will display an empty edit window:

```
PROGRAM HELLO  
CRT "Hello World!"  
STOP  
END
```

The program starts with a declaration that this is a program. This declaration is optional for programs but is required for external subroutines (`SUBROUTINE`), functions (`FUNCTION`), and classes (`CLASS`). If the name declared here differs from the name by which the program is stored in the file, then the compiler will give a warning message. While it is permitted for these names to be different, it is good practice to make them the same, and certainly makes it easier to debug applications.

The second line of the program does all the work. This prints the string "Hello World!" to the terminal.

The `STOP` command is not strictly necessary, but it is good practice to use them at the expected end of the executable code.

The `END` statement is also not strictly necessary, but the compiler will inform you if it is not present. The `END` statement marks the end of the program.

Close the `WED` editor, and save the program when prompted. In `WED`, you can also save the program by clicking on the save icon, or by selecting 'Save' or 'Save as' from the 'File' menu.

Now compile the program:

```
BASIC BP HELLO  
Compiling BP HELLO  
0 error(s)  
Compiled 1 program(s) with no errors
```

and run it:

```
RUN BP HELLO  
Hello World!
```

Finally, (assuming this is a program we want to keep), we will want to `CATALOGUE` the program. This lets us run the program simply by using the program name, rather than by using the `RUN` command.

```
CATALOGUE BP HELLO  
HELLO added to private catalogue
```

```
HELLO  
Hello World!
```

SD provides a number of options for cataloguing. The default method is private cataloguing as shown. .

By default, you will need to recatalogue the program every time it is compiled. One way to get around this it to use a compiler directive that directs the compiler to automatically catalogue the program. The compiler directive can be placed into each program individually or placed in a \$BASIC.OPTIONS item so that it applies to all programs within a file, or all programs within an account.

In this book, the compiler directive will be placed in each program. This makes our intentions explicit to anyone who subsequently edits the program.

Note that the commands invoking the editors have the same basic structure as virtually any other *SD* command:

```
command filename itemname
```

Therefore, if you are using an editor other than *SDEdit*, simply substitute the name of the editor you are using:

```
SED BP HELLO
```

Likewise, if you are using a different filename:

```
SDEdit PROGS HELLO
```

Statements, variables, tokens, constants, and operators

Statements

A program is made up of a series of statements which collectively tell the computer what to do. Generally, programs should be written with one statement per line – although multiple statements may be written on a single line if they are separated by a semi-colon.

```
statement 1 ; statement 2 ; statement 3
```

Statements beginning with an asterisk (*), an exclamation mark (!), or the keyword *REMARK* (or *REM*) are considered to be comments. Comments are often appended to the end of an active statement by using a semi-colon to denote a new statement, followed by the comment marker.

As noted above, programs are normally written with one statement per line. However, there are always exceptions:

- Some statements cover several lines
- Comments are often placed as a second statement on a line
- Sometimes, it is more convenient to place related statements together on a single line.

An example of all three of these possibilities is shown below:

```
BEGIN CASE
CASE papersize = 'A4';      papersizeindex = 9
CASE papersize = 'LETTER'; papersizeindex = 1
CASE papersize = 'EXECUTIVE'; papersizeindex = 7
CASE papersize = 'LEGAL';  papersizeindex = 5
CASE papersize = 'B5';     papersizeindex = 13
CASE papersize = 'ENV10';  papersizeindex = 15
CASE papersize = 'DL';     papersizeindex = 19
CASE 1;                    papersizeindex = 9; * A4 default
END CASE
```

This code fragment sets the variable *papersizeindex* based on the literal value of *papersize*.

- This whole block of statements makes up a CASE statement.
- There is a comment at the end of the line that starts with CASE 1.
- Each of the individual CASE lines has a second statement that assigns the *papersizeindex* variable.

Variables

SDBasic variables have relatively few naming restrictions.

- They must start with a letter
- They must not end with an underscore
- They consist of letters, digits, dollar signs, percentage signs, periods (full stops) and underscores.

The documentation provides a short list of reserved names which cannot be used as variable names. This means that it is technically possible to use the other keywords as variables. However, this is not a good idea as it will lead to confusion over whether you are using a variable name or a keyword.

There is no restriction on the length of variable names – although for practical purposes you should avoid very long names.

SDBasic variables can hold data of any type, and can change their type during the course of program execution. Therefore, we could write:

```
temp = '05'      ; * Define temp as string variable
CRT temp
temp = temp + 0 ; * Redefine as numeric
CRT temp
temp = 'X':temp ; * Redefine as string
CRT temp
```

and the output from this program would be:

05
5
X5

Variables should be declared before they are used. However, unlike some languages, this declaration does not need to be formal – it is simply an assignment with an initial value:

```
temp = 0
```

If this assignment does not take place, the program will abort at runtime with a variable not assigned error. For example:

```
PROGRAM TEST
FOR ii = 1 TO 10
  jj += 1
  CRT jj
NEXT ii
END

BASIC BP TEST
RUN BP TEST
000000B1: Unassigned variable JJ at line 3 of D:\SD\SDINTRO\BP.OUT\TEST
```

In this example, the variable *jj* had no value when the program encountered it for the first time.

Tokens

Tokens are predefined language elements that you use to represent a value in the programming environment. They aren't variables because you (usually) cannot assign new values to them, nor are they constants because they may change.

In *SD* (as with other multi-value environments), tokens are prefixed by the '@' symbol. The tokens dealing with multi-value delimiters have already been covered in Section 2.4.3. A few of the other frequently used tokens are shown in the table below:

Token	Meaning
@TRUE	A logical true value – nominally 1
@FALSE	A logical false value – nominally 0
@LOGNAME OR @USER	The user's operating system logon name
@SENTENCE	The most recently executed <i>SD</i> sentence
@WHO	The current account

The full list of tokens (or @-variables) can be found in the *SD* help files under *SDBasic*.

Constants

Constants are language elements that are assigned values that never change. Further, if assignment of a new value is attempted, an error will result.

Constants are assigned using the EQUATE statement:

```
EQUATE pi TO 3.141592654
```

Thereafter, constants may be used within SDBasic expressions:

```
area = pi * PWR(radius, 2)
```

Multi-value Variables

SD is a multi-value database, and accordingly, variables used in the programming language may be multi-valued. Such variables may also be termed ‘dynamic arrays’ or delimited strings, and are denoted as:

```
varname<amc {,vmc {,svmc}} >
```

In this syntax, the angle brackets ‘<’ and ‘>’ indicate the use of a multi-valued variable. Multi-valued variables can have up to 3 dimensions corresponding to attributes¹⁵ (fields), values, and sub-values. The following are valid multi-value variables:

```
abc<3>
def<2, 1>
ghi<ii, jj, kk>
```

The final example uses variables to denote the index positions of each dimension of the variable.

If you wish to append data to the array, you can use an index value of -1. For example:

```
abc<-1>
def<-1, 1>
ghi<3, 1, -1>
```

Note that you do not need to explicitly set the dimensions of the variable before use. You can simply add dimensions, and add elements to each dimension as required:

```
score = ''
FOR die1 = 1 TO 6
  FOR die2 = 1 to 6
    score<die1, die2> = die1 + die2
  NEXT die2
NEXT die1
```

This creates a 2-dimensional dynamic array holding all the possible scores from rolling 2 dice. This could then be used to return the combined value of two die:

```
rollvalue = score<die1, die2>
```

Referencing positions in dynamic arrays is often done by the numeric position in the array. However, as dynamic arrays often hold data that has been read from a file, it is often useful to use an equated constant that gives you a better idea of what the data is in the variable. Consider the following statements:

15 They syntax description here uses abbreviations of amc, vmc, and svmc. These mean “attribute mark count”, “value mark count”, and “sub-value mark count”. These abbreviations are common in the PICK world.

```
rec<10> = rec<8> + rec<9>
```

```
sales.rec<SL.TOTAL> = sales.rec<SL.SUBTOTAL> + sales.rec<SL.VAT>
```

There are two points to take from these statements:

- Using equated constants has made the statement much more understandable – you don't have look up what data is held in positions 8, 9, and 10 in the dynamic array to understand the meaning of the statement
- Mistakes in programming logic are much easier to spot because you can read the purpose of the statement. For example, the first statement may have been written as:

```
rec<10> = rec<8> + rec<19>
```

The mistake is not immediately obvious. This mistake would be unlikely if equated constants had been used.

SD can create a set of equated constants for you from a file dictionary. The command for this is

GENERATE:

GENERATE XRATES

Type (D=dynamic array, M=matrix, or DM=both): **D**

Prefix for dynamic array tokens (excluding separator): **XR**

This creates an item in the BP file that can be included in programs:

CT BP XRATES.H

BP XRATES.H

01: * BP XRATES.H

02: * Generated from DICT XRATES at 18:16:42 on 16 May 2009

03:

04: equate XR.TWI to 1

05: equate XR.TWI.CNV to "MR1,Z"

06: equate XR.USD to 2

07: equate XR.USD.CNV to "MR4,Z"

08: equate XR.GBP to 3

09: equate XR.GBP.CNV to "MR4,Z"

10: equate XR.AUD to 4

11: equate XR.AUD.CNV to "MR4,Z"

12: equate XR.JPY to 5

13: equate XR.JPY.CNV to "MR2,Z"

14: equate XR.EUR to 6

15: equate XR.EUR.CNV to "MR4,Z"

16: equate XR.GDM to 7

17: equate XR.GDM.CNV to "MR4,Z"

Note this has created constants not only for the field numbers in the record, but also the conversion codes used to convert data from internal to external format.

Operators

SD has the usual set of operators for writing expressions and relational conditions. In addition, it has a set of substring extraction operators, a pattern matching operator, and a set of alternative relational operators.

Substring extraction

SD uses square brackets ([and]) to denote substring extraction as follows:

```
substring = string[startpos, numchars]
```

For example:

```
st = 'AbcDefGhi'  
ss = st[4,3]
```

In the above program fragment, the substring 'Def' is assigned to the variable *ss*.

To extract the last *n* characters, use:

```
substring = string[n]
```

For example:

```
st = 'AbcDefGhi'  
ss = st[3]
```

In this example, the substring 'Ghi' is assigned to the variable *ss*.

In some other languages, this substring extraction functionality is provided by the LEFT, MID, and RIGHT functions.

Pattern matching

Pattern matching is often used in the context of an IF statement:

```
IF var MATCHES pattern THEN ...
```

For example:

```
xx = '(04) 456 7890'  
IF xx MATCHES '"("2N") "3N" "4N' OR xx MATCHES '"("2N") "3N"- "4N' THEN
```

The above program fragment matches the test string to see if it is a phone number. Note, however, that not all phone numbers will match the test patterns. Clearly, some thought needs to be put into the appropriate tests for phone numbers.

For more information on how to define the test patterns, search for MATCHES in the online help.

Alternative relational operators

Most languages have a set of mathematical symbols for use in relational comparisons. *SD* has a set of mnemonic codes that act as synonyms for these mathematical symbols. These are shown in the table below:

Symbol	Alternatives
<	LT
>	GT
=	EQ
#	NE <> ><
<=	LE =< #>
>=	GE => #<
MATCHES	MATCH
AND	&
OR	!

For example:

```
IF xx LT 1 THEN xx = 1
```

Assignment

Simple assignment takes the form:

```
var = value
var = expression
```

Assignment shortcuts

In common with some other languages, *SD* offers shortcuts for repeated operations. For example:

```
xx = xx + 1
```

could be written as:

```
xx += 1
```

This not only saves keystrokes in typing, but is actually more efficient in execution. These shortcuts take the general form:

```
var shortcut value
```

The following set of assignment shortcuts are available:

```
+= Add expression to the original value
-+ Subtract expression from the original value
*= Multiply original value by expression
/= Divide original value by expression
:= Concatenate expression as a string to the original value
```

For example:

```
outputstring := thisline:CR:LF
```

This takes the current value of `outputstring` and appends the string value of `thisline` and a carriage return/line feed sequence.

Note this also introduces the use of the colon (:) as the means of concatenating two strings together. For example:

```
newstring = string1:string2
```

Whitespace can be introduced into the statement to make the operators more visible without changing the functionality of the statement. For example:

```
newstring = string1 : string2
```

This separation makes each component of the statement more visible, and can aid with understanding.

Substring assignment

Substring extraction was outlined in the section on operators above. Similar techniques can be used to assign values to substrings:

```
var[startpos, numchars] = expression
```

For example:

```
st = 'Abcdefghi'  
st[4,3] = '123'
```

The new value of *st* would be: *Abc123ghi*

Now consider the following example:

```
st = 'Abcdefghi'  
st[4,3] = '12'
```

In this case, the new substring is shorter than the existing substring. *SD*'s behaviour (and the resulting value of *st*) here is dependent on the `$MODE` settings used to control the compiler. See the online help for more information.

A shortcut method is available to assign the trailing characters of a string:

```
var[numchars] = expression
```

For example:

```
st = 'Abcdefghi'  
st[3] = '123'
```

The new value of *st* would be: *Abcdef123*

Finally, *SD* has another form of substring assignment particularly suited to use with delimited strings:

```
var[delimiter, firstgroup, numberofgroups] = expression
```

For example:

```
key = '1234*DEF*14996'  
key['*', 3, 1] = DATE()
```

This would replace the 14996 part of the key with the internal value of the current date. For the 14th of April, 2009, this would make the key value: *1234*DEF*15080*

Null values

In common with the most other multi-value databases, *SD* does not have a special character representing the null value. For practical purposes, an empty string is considered a null value.

A simple program

The best way to illustrate the statements and functions available in *SD* is to show them in operation. Therefore, we will now consider a short program and examine how it works.

Part 1 of '*Getting Started in SD*' used several files to illustrate how to use *SDQuery*. These files contained interest and exchange rate for New Zealand. Building on this base, an appropriate first program will allow this data to be viewed.

Program

```
PROGRAM SHOW.XRATES
*****
* Program to display exchange rates for a year nominated by the user.
*
$CATALOGUE

PROMPT ''
OPEN 'XRATES' TO xrates ELSE STOP 201, 'Xrates'
OPEN 'DICT','XRATES' TO xrates.dict ELSE STOP 201, 'Dict Xrates'

! Define constants

$INCLUDE SYSCOM KEYS.H
$INCLUDE BP XRATES.H

! Read conversions and headings from dictionary

dnames = 'TWI,USD,GBP,AUD,JPY,EUR'
CONVERT ',' TO @AM IN dnames

cnvs = ''
hdgs = ''
FOR cc = XR.TWI TO XR.EUR
  dictname = dnames<cc>
  READ drec FROM xrates.dict,dictname THEN
    conversion = drec<3>
    IF conversion EQ '' THEN conversion = 'MR0,Z'
    cnvs<cc> = conversion
    hdgs<cc> = drec<4>
  END
NEXT cc

! Page heading

CRT @(IT$CS):
CRT @(00,00):'Display exchange rates'
CRT @(00,01):'=====

! Main loop

LOOP
CRT @(00,03):@(IT$CLE0S):'Enter year: ':
INPUT yyyy:
IF (yyyy EQ '') OR (UPCASE(yyyy) EQ 'X') THEN EXIT

IF NOT(yyyy MATCHES '4N') THEN
  CRT @(18,03):'Invalid year':
  INPUT pause,1:
  CONTINUE
END

! Read data from file and display

first = @TRUE
FOR mm = 1 TO 12
  xid = yyyy:mm 'R%'
  READ xrec FROM xrates, xid THEN
    IF first THEN
      CRT @(00,05):'Month':
      FOR cc = XR.TWI TO XR.EUR
        CRT hdgs<cc> 'R#12':
      NEXT cc
      CRT
      first = @FALSE
    END

    xdate = '01/' : mm : '/' : yyyy
    CRT OCONV(ICONV(xdate, 'D'), 'DMAL[3]') 'L#5':
    FOR cc = XR.TWI TO XR.EUR
      CRT OCONV(xrec<cc>, cnvs<cc>) 'R#12':
    NEXT cc
    CRT
  END
NEXT mm
```

```

IF NOT(first) THEN
  CRT
  CRT 'Press any key to continue ':
END ELSE
  CRT @(00,05):'No data on file for year ':yyyy:
END
INPUT pause,1:
REPEAT

STOP
END

```

Analysis

This program asks the user to enter a year. It then displays the exchange rate data on file for that year. The data is displayed until the user presses enter, and then the program returns to the prompt for a year. The user exits the program by pressing enter at the year prompt. Sample output is shown below:

```

Display exchange rates
=====

Enter year: 2004

Month  TWI      USD      GBP      AUD      JPY      EUR
Jan    66.4    0.6724  0.3690  0.8728  71.55   0.5332
Feb    68.0    0.6916  0.3703  0.8891  73.68   0.5473
Mar    66.3    0.6614  0.3616  0.8811  71.90   0.5388
Apr    64.7    0.6419  0.3556  0.8622  68.94   0.5350
May    63.1    0.6156  0.3445  0.8731  69.00   0.5127
Jun    64.2    0.6293  0.3440  0.9058  68.81   0.5184
Jul    65.4    0.6466  0.3508  0.9030  70.67   0.5268
Aug    66.5    0.6542  0.3594  0.9209  72.19   0.5370
Sep    67.1    0.6588  0.3674  0.9384  72.49   0.5392
Oct    68.5    0.6825  0.3782  0.9326  74.47   0.5471
Nov    68.3    0.6993  0.3764  0.9084  73.29   0.5387
Dec    69.0    0.7142  0.3702  0.9315  74.18   0.5337

Press any key to continue

```

Let's step through the program to see how it works:

The first functional line is `$CATALOGUE`. This is a compiler directive rather than a statement, and tells the compiler to automatically catalogue the program in the private catalogue every time it is compiled.

`PROMPT "` sets the prompt character to nothing. By default, *SD* displays a '?' character whenever the program reaches an `INPUT` statement. By setting it to null, the developer has full control over how the program appears to the user.

The two `OPEN` statements open the `XRATES` data file and dictionary. This lets the program read from these files later.

The two `$INCLUDE` statements include code fragments from elsewhere on the system. The first of these is the keys definition file included with *SD*. The second is the file definition item we created for the `XRATES` file using the `GENERATE` command in the section on Constant. By including these files, we can use mnemonic codes for file references and system functions rather than numbers.

The next section reads the column headings and conversions to apply to the data from the file dictionary. First, we create a dynamic array that contains the ID's of the dictionary items. This is done in two steps – firstly by creating a string variable containing the ID's, and then by converting the string

to a dynamic array by changing the commas to attribute marks. This approach is used because it is easier than writing a line such as:

```
dnames = 'TWI' :@AM: 'USD' :@AM: 'GBP' :@AM: 'AUD' :@AM: 'JPY' :@AM: 'EUR'
```

Next, the conversions and column headings for each of the columns are read from the dictionary. These conversions transform the stored data from its internal format to its external format. While these conversions were also defined in the include file created using the `GENERATE` command, reading them from the dictionary puts them into a more general structure which makes them easier to use.

The screen is then cleared, and some page headings put in place.

The rest of the program is encompassed by the main loop. This starts with a statement that clears the screen from line 3 onwards, and displays a prompt asking the user to enter a year. This year is then tested to see if it matches a 4 digit pattern. If the year does not match this pattern, then an error message is displayed, and the loop is started again.

If the year does contain 4 digits, then the program will attempt to read data for that year. Note that this will happen even if the year entered is nonsensical – such as 0000 or 2050. However, the read is structured so that all years are handled appropriately, whether they contain data or not.

An attempt is made to read the data for each month of the year. If data for this month is found on file, then the data is displayed. If this is the first month found for the year, then the column headings are written to the screen first. This means that if no data are found for the year, then no column headings are displayed.

Each line of data is displayed by first calculating the month name. Once the month is displayed, then the program loops through the record converting the data to external format, and displaying it.

Finally, a message is displayed, either asking the user to press any key to continue, or informing them that there was no data on file for that year. This message is displayed until the user presses enter, when control returns to the top of the loop.

Detail points

The program uses defined constants wherever possible to avoid incorrect numbers being accidentally used. Example of this are:

```
FOR cc = XR.TWI TO XR.EUR
```

```
CRT @(IT$CS):
```

The constants `XR.TWI` and `XR.EUR` were defined in the `XRATES.H` include item, and evaluate to values of 1 and 6 respectively. Therefore, the `FOR` statement evaluates to:

```
FOR cc = 1 TO 6
```

Likewise, the `CRT` statement evaluates to:

```
CRT @(-1):
```

This expression is used to clear the screen. Similarly, the line:

```
CRT @(00,03):@(IT$CLEOS):'Enter year: ':
```

firstly, positions the cursor, then clears the screen from the position onwards using the @(-3) function.

The constants used in these screen functions are defined in the KEYS.H item in the SYSCOM file. This item contains many more constants for use within *SD*.

These CRT statements are terminated by a colon. A colon in an expression is used to concatenate two or more elements together. When used at the end of a CRT statement, it acts to hold the cursor at that position. If the colon were not present at the end of the “Enter year” statement, then an automatic end of line sequence would be executed, causing the cursor to go to the left hand edge of the following line.

A number of lines have formatting expressions:

```
xid = yyyy:mm 'R%'
```

```
CRT hdgs<<cc> 'R#12':
```

In the first of these, the format expression 'R%' forces the variable *mm* to be formatted right-justified in a field 2 characters wide and padded with zeroes. Therefore, a value of *mm* of 1 (i.e. month 1 or January) would be formatted as 01. This is concatenated to the 4 digit year (say 2009) to produce an exchange rate identifier of 200901. This identifier is then used to read the data for that month from the exchange rates file.

The second use of the format expression is to format the output of the column headings (and later of the rows of data). The 'R#12' expression means format the data right-justified in a field of 12 blank characters.

The printing of the month name is accomplished by the following two lines:

```
xdate = '01/':mm:':':yyyy  
CRT OCONV(ICONV(xdate, 'D'), 'DMAL[3]') 'L#5':
```

The first line is clear enough. This creates an international format date string from the month number and year. Therefore, for a month number of 1 and a year of 2009, *xdate* would be '01/1/2009'. If you are in North America, you would need to change this date string, or specify a European date conversion in the following statement.

The second line is a little more complicated, but is really just three operations carried out in a single line. First, the date string is converted to an internal date:

```
ICONV(xdate, 'D')
```

So, '01/01/2009' would be converted to 15128. (Type in: DATE INTERNAL 01/01/09 from the command prompt).

The next part gets a month representation from that date:

```
OCONV(15128, 'DMAL[3]')
```

The 'DMA' portion of the conversion expression returns the month name as 'JANUARY'. The 'L' converts this to Title case (January), and the '[3]' truncates this to 3 characters (Jan).

Finally, this is displayed left-justified in a field of 5 blank characters:

```
CRT 'Jan' 'L#5':
```

Note the presence of the colon to keep the cursor on the current output line.

The final details involve printing a line of data:

```
FOR cc = XR.TWI TO XR.EUR  
  CRT OCONV(xrec<cc>, cnvs<cc>) 'R#12':  
NEXT cc  
CRT
```

This code fragment displays each of the columns in a field of 12 characters wide on the screen. Each field is placed immediately to the right of the preceding field because the cursor is held on the line by the colon following the format expression. Once the loop is complete for each row, the cursor is still held on the output line. The CRT statement on its own outside the loop moves the cursor to the next line to print a new month's data.

The actual expression used to display the data is also interesting. This is easier to understand if we substitute the variable *cc* for one of its actual values:

```
CRT OCONV(xrec<XR.TWI>, cnvs<XR.TWI>) 'R#12':
```

This expression says take the TWI value from the exchange rate record and apply the output conversion defined for the TWI field, then display it right-justified in a field of 12 spaces. If we go back to the sample output displayed above, we can see that the TWI for January 2004 was 66.4 – this is the output format. This was actually stored as 664, and the output conversion of 'MR1,Z' converted this internal representation of 664 to an external representation of 66.4. To check the actual conversion used by TWI, check the conversion stored in the dictionary:

```
CT DICT XRATES TWI
```

The conversion appears on line 3 of the dictionary. We read these output conversions from the dictionary at the start of the program to fill the *cnvs* dynamic array. Because the structure of the *cnvs* dynamic array matches that of the *xrec* dynamic array that holds the exchange rate, we could use these in a matched fashion in the display loop.

It is worthwhile considering how we could have used the conversions defined in the \$INCLUDE statement to display the same data. To have used those conversions, we would have had to refer to them by their name. That would have meant the display statement would have been something like:

```
CRT OCONV(xrec<XR.TWI>, XR.TWI.CNV>) 'R#12':  
CRT OCONV(xrec<XR.USD>, XR.USD.CNV>) 'R#12':  
CRT OCONV(xrec<XR.GBP>, XR.GBP.CNV>) 'R#12':  
etc
```

So, by putting the conversions in a dynamic array, we were able to make the display portion of the program much more compact. Of course, we could have accomplished this by creating the *cnvs* dynamic array from the defined conversions rather than reading from the dictionary:

```
cnvs = ''
cnvs<XR.TWI> = XR.TWI.CNV
cnvs<XR.USD> = XR.USD.CNV
cnvs<XR.GBP> = XR.GBP.CNV
etc
```

In fact, this is probably a better method of loading the conversions than reading from the dictionary. The reason for this is that *SD* supports dictionary structures used in other multi-value databases that have the conversion in a different place in the dictionary. Further, even if we adapted the read section to get the conversion from the correct position, we would also need to check that the value did not contain some other processing codes. Therefore, our dictionary read section would become something like:

```
cnvs = ''
hdgs = ''
FOR cc = XR.TWI TO XR.EUR
  dictname = dnames<cc>
  READ drec FROM xrates.dict,dictname THEN
    dtype = drec<1>[1,1]
    BEGIN CASE
      CASE (dtype EQ 'D') OR (dtype EQ 'I') OR (dtype EQ 'C')
        conversion = drec<3>
        hdgs<cc> = drec<4>
      CASE (dtype EQ 'A') OR (dtype EQ 'S')
        conversion = drec<7>
        hdgs<cc> = drec<3>
      CASE 1
        conversion = ''
    END CASE
    dc = DCOUNT(conversion<1>, @VM)
    IF dc GT 1 THEN
      conversion = conversion<dc>
    END
    IF conversion = '' THEN conversion = 'MR0,Z'
    cnvs<cc> = conversion
  END
NEXT cc
```

This code fragment gets the heading and conversion from the appropriate position for each dictionary type. Note the following points:

- Some dictionary items (usually A and S types) may have multi-valued conversion fields. The code therefore checks for the presence of value marks and gets the last value as the conversion code.
- If the dictionary item is not one of the specified types, then the conversion is explicitly set to a null value. If we didn't do this, then the variable 'conversion' would still contain the conversion from the previous dictionary item (or would be undefined if this was the first loop).
- Headings are only set for the specified dictionary types. If the dictionary is of some other type, then no heading is set.

The alternative to this code is to use the *GENERATE* command to create an include file containing the conversions. You can then check that include file to make sure the conversions are valid, and edit them if necessary.

Other improvements that could be made include reading the display width for each column from the dictionary (field 5 of a D, I, or C type dictionary), and allowing for multi-line column headings.

Overall, this is a fairly simple program. However, it introduces many of the basic elements that are present in larger programs – data input, validation, reading data from a file, formatting data, and outputting formatted data to an output device. All that is really missing is writing data to a file.

Some useful functions and statements

There are a number of functions that are frequently used in multi-value databases. It is useful to outline some of these here so that you will understand them as we cover other programming structures.

CRT and DISPLAY

CRT and DISPLAY are the same function. In this book, CRT will be used, but in other programs you may find DISPLAY being used.

As its name suggests, CRT outputs data to the screen. Its syntax is:

```
CRT printitem {,printitem}
```

The printitem may be a variable, expression, or a literal string. If there are multiple printitems separated by commas, the commas are replaced by TAB characters to produce a simple formatted display.

You can include a cursor positioning command as part of the CRT statement:

```
CRT @(col,line)
```

```
CRT @(col)
```

Line and column numbering starts at zero, so @(0,0) will position the cursor at the top left of the screen. The second version shown will position at 'col' on the current line.

The CRT @(x) function also accepts negative arguments for terminal control purposes. Some useful examples of these include:

```
CRT @(-1)          Clear screen
CRT @(-3)          Clear to end of screen
CRT @(-4)          Clear to end of line
```

Tokens to use with the CRT function are contained in the KEYS.H include item in the SYSCOM file. Tokens are useful because they make it less likely that you will use an incorrect numeric value, and they make the code easier to read. The tokenised equivalents of the three CRT statements shown above are:

```
CRT @(IT$CS)
CRT @(IT$CLEOS)
CRT @(IT$CLEOL)
```

To use these tokens, you need to \$INCLUDE the KEYS.H item in your program.

Note that the `CRT` function issues an automatic end of line sequence, so if you want to hold the cursor at that position, you need to suppress this sequence. This is done by appending a colon (`:`) to the command:

```
CRT @(20,05):
```

It is common for a `CRT` statement to contain an expression which builds a formatted string for display:

```
CRT @(sx,sy):qty<mathno> 'R#12Z':
```

The above statement outputs the value contained in the `mathno` attribute of the `qty` variable at position (`sx`, `sy`) on the screen, displayed right-justified in a field of 12 spaces. See the online help for more information on format specifications.

PRINT

`PRINT` is closely related to the `CRT` function. However, whereas the `CRT` function always outputs to the screen, `PRINT` will output to an output device – whether that is the screen or a print unit (printer). Its syntax is:

```
PRINT {ON printunit} printitem {,printitem}
```

Output is directed as follows:

- If a print unit is specified, then output will be directed to that print unit
- If the print unit specified is `-1`, then output will go to the screen
- Print unit `0` (the default) is switchable between the screen and the print unit by use of the `PRINTER ON/OFF` statement
- Printunit may be a number between `0` and `255`. The characteristics of the print unit may be set using the `SETPTR` command from the `SD` command environment, or the `SETPU` statement from within `SDBasic`.

```
PRINT "This text will go to the screen"  
PRINTER ON  
PRINT "This text will go to the print unit"  
PRINT ON -1 "This text will go to the screen"  
PRINTER OFF
```

As with the `CRT` statement, you can include cursor positioning commands with the `PRINT` statement – but these will have no effect if output is directed to a print unit.

INPUT

`INPUT` accepts input from the keyboard or data queue and stores that input in a variable:

```
INPUT variable {,length}
```

`INPUT` accepts a number of other parameters not shown above – see the online help for more details.

If length is specified, then input of characters is automatically terminated when that number of characters has been input:

```
INPUT pause,1
```

A more sophisticated version of INPUT is the INPUT @(x,y) statement. This version allows cursor positioning, formatting of the displayed/entered variable, and various modes of editing – see the online help for more details.

DCOUNT

DCOUNT is used to count the number of components in a delimited string. As multi-value variables are simply delimited strings, it is frequently used to count the number fields, values, or sub-values in a variable. Its syntax is:

```
number = DCOUNT(string, delimiter)
```

For example:

```
menucnt = DCOUNT(menuitems<1>, @VM)
```

This example counts the number of values in the first field of the variable *menuitems*.

There is a closely related function named COUNT which simply counts the delimiters:

```
number = COUNT(string, delimiter)
```

Usually, DCOUNT returns a number that is one greater than that COUNT.

FIELD

FIELD extracts a component of a delimited string. This isn't usually used for extracting fields, values, and sub-values as the language provides for direct extraction and assignment of these elements. However, we are frequently confronted by strings delimited by other characters.

For example, a compound key may have the structure:

```
plant * model no * date  
e.g. 123*A-4567-IJK*15096
```

Each of these components may have variable length (although date should be stable at 5 characters for some time to come), so we cannot extract by position. This is where we use the FIELD function. This has the following syntax:

```
component = FIELD(string, delimiter, occurrence {, count})
```

Therefore to extract each of the components of the variable 'key', we would use:

```
plant = FIELD(key, '*', 1)   returns '123'  
modelno = FIELD(key, '*', 2) returns 'A-4567-IJK'  
pdate = FIELD(key, '*', 3)  returns 15096
```

Note that the strings returned by the field statement exclude the delimiter – unless more than one field is returned:

```
plantmodel = FIELD(key, '*', 1, 2)      returns '123*A-4567-IJK'
```

Similar functionality can be achieved by using a group extraction conversion in an OCONV function:

```
PROGRAM TEST
key = '123*A-4567-IJK*15096'
plant = FIELD(key, '*', 1)
modelno = FIELD(key, '*', 2)
pdate = FIELD(key, '*', 3)
plantmodel = FIELD(key, '*', 1, 2)
pm2 = OCONV(key, 'G0*2')

CRT 'Plant = ':plant
CRT 'Model = ':modelno
CRT 'Pdate = ':pdate
CRT 'PlantModel = ':plantmodel
CRT 'PM2 = ':pm2

END
```

```
RUN BP TEST
Plant = 123
Model = A-4567-IJK
Pdate = 15096
PlantModel = 123*A-4567-IJK
PM2 = 123*A-4567-IJK
```

FIELD has two associated functions – COL1() and COL2() – which return the character positions immediately prior to (COL1()) and immediately following (COL2()) the extracted string. These are useful for extracting the strings prior to and/or following the extracted field:

```
key = '123*A-4567-IJK*15096'
modelno = FIELD(key, '*', 2)
priorstring = key[1,COL1()]
followstring = key[COL2(), LEN(key)]
```

See the online help for more information.

ICONV / OCONV

ICONV and OCONV are transformation functions – ICONV converts data to internal format, while OCONV converts data to external format. The 'I' and the 'O' stand for input and output, representing the type of conversion that takes place at these times. An input conversion will be done at the time of data input, and converts data from external to internal format. An output conversion will be done prior to display, and converts data from internal to external format.

Dates and times are obvious examples of data with differing input and output representations, but any data may have conversions applied using these functions. Some examples are shown below:

Conversion	Result
x = OCONV(15100, 'D')	04 MAY 2009
x = ICONV('4 MAY 2009', 'D')	15100
x = OCONV(32000, 'MTS')	08:53:20
x = ICONV('14:20', 'MTS')	51600
x = OCONV('SAMPLE TEXT', 'MCL')	sample text
x = OCONV('sample text', 'MCU')	SAMPLE TEXT
x = OCONV('SAMPLE TEXT', 'MCT')	Sample Text
x = ICONV(123.456, 'MR33')	123456
x = OCONV(123456, 'MR22,\$')	\$1,234.56

See 'Conversion Codes' in the online help for more information on possible conversions.

LOCATE

LOCATE is used for finding a value within a delimited string. LOCATE is frequently used where a variable contains sets of related data.

The basic concepts of the LOCATE statement are:

- A search for a string is made through the specified part of a delimited variable returning:
 - whether the string was found
 - the position in the variable that the variable occupies, or would occupy if it existed
- THEN OR ELSE clauses are executed depending on whether the string was found.

SD allows several variants for the syntax of the LOCATE statement. The variant shown here is the *UniVerse* version. This requires a MODE setting in the program:

```
$MODE UV.LOCATE
LOCATE string IN dyn.array {<field {,value}>} {BY order} SETTING pos {THEN statements } {ELSE
statements}
```

The \$MODE statement need only be declared once in the program. Alternatively, it can be specified in a \$BASIC.OPTIONS item in the programs file, or in the VOC of the account.

Say we want to search through the days sales and return the value of sales by country.:

```
OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
countries = ''
SELECT sales
eof = @FALSE
LOOP
  READNEXT sales.id ELSE eof = @TRUE
UNTIL eof DO
  READ sales.vec FROM sales, sales.id ELSE sales.vec = ''
  sales.ctry = sales.vec<SL.CTRY>
  IF sales.ctry = '' THEN sales.ctry = 'GB'
  sales.value = sales.vec<SL.VALUE>
  IF sales.ctry AND sales.value THEN
    LOCATE sales.ctry IN countries<SS.CTRY> BY 'AL' SETTING cpos THEN
      countries<SS.CTRYVALUE, cpos> += sales.value
  END ELSE
    INS sales.ctry BEFORE countries<SS.CTRY, cpos>
    INS sales.value BEFORE countries<SS.CTRYVALUE, cpos>
  END
END
REPEAT
```

This code fragment builds the dynamic array *countries* by searching for the country identifier in the relevant field of the dynamic array. This search uses an ascending left-justified sort order (i.e. alphabetic). If the country already exists, then the position of the country in the field is returned in *cpos*, and the value of the sale is added to the existing sales values for that country in the appropriate field. If the country does not exist, then *cpos* contains the location within the dynamic array where it should be inserted, and both the country identifier and the sales value are inserted into the dynamic array.

Note that for this code fragment to work, there needs to be an `$INCLUDE` statement that defines the `SL.CTRY`, `SL.VALUE`, `SS.CTRY` and `SS.CTRYVALUE` constants.

`LOCATE` can also be used without a sort order. In this case, the `LOCATE` portion of the above code fragment would look like:

```
LOCATE sales.ctrly IN countries<SS.CTRY> SETTING cpos ELSE
  countries<SS.CTRY, cpos> = sales.ctrly
END
countries<SS.CTRYVALUE, cpos> += sales.value
```

This code is a bit simpler because it doesn't have to insert values into the dynamic array. If the country is found in the dynamic array, then all that needs to happen is to add the sales value to the appropriate position. If the country is not found, then the value returned by *cpos* is at the end of the existing data – therefore, you can simply assign the country to that position and add the sales data. Because the sales data is added to a position in the dynamic array whether the country is found or not, this is removed from the locate statement logic and made an unconditional statement.

Note the following points about the use of `LOCATE`:

- Searching within a dynamic array will slow down as the array size gets larger. For this reason, avoid building large lists if possible.
- A sorted list will be more efficient than an unsorted list. This is because once the location of the target is found (whether or not it currently exists), searching will stop. However, with an unsorted dynamic array, searching needs to continue right through the list to determine whether it is present in the list.
- `LOCATE` finds an entire value within the dynamic array. If you want to find part of a string, then you need to use a different function.

Now, while the above example usage of `LOCATE` may be a reasonable demonstration of using `LOCATE`, it is not an efficient piece of code. This is because the `LOCATE` will run for every item in the sales file. This is quite unnecessary – and inefficient. Consider the following alternative:

```

OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
EXECUTE \SSELECT SALES BY CTRY\
countries = ''
sales.value = 0
sales.ctry.prev = ''
eof = @FALSE
LOOP
  READNEXT sales.id ELSE eof = @TRUE
UNTIL eof DO
  READ sales.vec FROM sales, sales.id THEN
    sales.ctry = sales.vec<SL.CTRY>
    IF sales.ctry = '' THEN sales.ctry = 'GB'
    IF sales.ctry NE sales.ctry.prev THEN GOSUB addctry
    sales.value += sales.vec<SL.VALUE>
  END
REPEAT

IF sales.value THEN GOSUB addctry
*
RETURN

addctry:
IF sales.value THEN
  countries<SS.CTRY, -1> = sales.ctry.prev
  countries<SS.CTRYVALUE, -1> = sales.value
END
sales.value = 0
sales.ctry.prev = sales.ctry
*
RETURN

```

This code fragment does not use LOCATE at all, but will build the same dynamic array of sales by country. The code works by sorting the data first, and then loops through the data accumulating the sales figures. The accumulated data is only added to the dynamic array when the country changes, or on final exit from the loop.

Now, say the dynamic array that is created by this code is saved to a sales summary file using a key of the internal date value. The following code fragment shows LOCATE being used to query the item and return the value of sales for a given country:

```

PROMPT ''
OPEN 'SALES.SUMMARY' TO sales.summary ELSE STOP 201, 'Sales.summary'
LOOP
  CRT 'Enter date: ':
  INPUT sdate
  IF (sdate = '') OR (UPCASE(sdate) = 'X') THEN EXIT
  CRT 'Enter country: ':
  INPUT ctry
  IF (ctry = '') OR (UPCASE(ctry) = 'X') THEN EXIT

  idate = ICONV(sdate, 'D')
  IF (idate = '') OR NOT(NUM(idate)) THEN
    CRT 'Invalid date'
    CONTINUE
  END
  sdate = OCONV(OCONV(idate, 'D'), 'MCT')
  READ sales.vec FROM sales.summary, idate ELSE
    CRT 'Sales data for ':sdate:' not on file'
    CONTINUE
  END

  LOCATE ctry IN sales.vec<SS.CTRY> SETTING cpos THEN
    sales.value = sales.vec<SS.CTRYVALUE, cpos>
  END ELSE
    sales.value = 0
  END
  sdate = OCONV(OCONV(idate, 'D'), 'MCT')
  CRT 'Sales for ':sdate:' were ':OCONV(sales.value, 'MR2,')
REPEAT

```

This program gets a date and a country from the user, then reads the summary item for that date. If no data exists for that date, then an error message is displayed, and the CONTINUE statement causes the program to start another LOOP. If data is found, then the country is located within the dynamic array, and the matching sales value returned. Finally, the sales value for the country for that date is displayed.

Note that the date value has some interesting conversions in this program. It is initially converted to an internal format, then the internal date value has a double output conversion applied to it. The first of these conversions converts the internal value to a standard date string (e.g. 08 MAR 2010), while the second converts this to a mixed case string (08 Mar 2010). This ensures consistency of display of date values.

As LOCATE is such an important statement within the multi-value databases, we'll give another example of its usage:

Say we have a banking application. Account balances are held in file ACCBALS. This file records the account balance of an account on any day that a transaction occurs. In addition, an account balance will be stored in the file on the last day of the month regardless of whether any transaction has occurred during the month. Our problem is: How do we efficiently find the account balance on any given day?

An account that is used daily will have as many account balances stored as there are (working) days in the month. An account that has no transactions during the month will only have a single account balance stored for the month.

The brute force method of getting the account balance is simply to attempt to read the account balance for the given day. If the item doesn't exist, then step back a day at a time until an account balance is found. The problem here is that we may need to attempt 30 reads from the file before we find the balance. This is not efficient.

A more efficient method involves indexing the monthly transactions. Let's set this out in a bit more detail:

The ACCBALS file has the following structure:

ID:	account*date	e.g.	12345678*15317
F1:	account balance	e.g.	1534682

The date in the account balance is 7 December, 2009. We know there will also be an item in the file with an ID of 12345678*15310 (being 30 November, 2009).

We also have an index file named ACCBALS.NDX. This has the following structure:

ID:	account*yyyymm	e.g.	12345678*200912
F1:	mv-list of dates	e.g.	15317]15314]15310

Note that the last date in the list is actually the November month-end date. The closing square brackets (]) represent value marks. The three dates shown are 7 December, 4 December, and 30 November.

When a transaction occurs, the application maintains the ACCBALS.NDX file, recording the dates on which new balances are written to the ACCBALS file. The month-end process has to create the index item for the next month and seed it with the date of the month-end just past.

Now, let's see how to use this structure to obtain the account balance:

```
PROMPT ''
OPEN 'ACCBALS' TO accbals ELSE STOP 201, 'Accbals'
OPEN 'ACCBALS.NDX' TO accbals.ndx ELSE STOP 201, 'Accbals.Ndx'
LOOP
  CRT 'Enter date: ':
  INPUT sdate
  IF (sdate = '') OR (UPCASE(sdate) = 'X') THEN EXIT
  idate = ICONV(sdate, 'D')
  IF (idate = '') OR NOT(NUM(idate)) THEN
    CRT 'Invalid date'
    CONTINUE
  END

  CRT 'Enter account: ':
  INPUT account
  IF (account = '') OR (UPCASE(account) = 'X') THEN EXIT

  ndxid = account:*':0CONV(idate, 'DY'):0CONV(idate, 'DM') 'R%%'
  READV ndx.rec FROM accbals.ndx, ndxid ELSE
    CRT 'This account was not open on this date'
    CONTINUE
  END

  LOCATE idate IN ndx.rec<1> BY 'DR' SETTING datepos ELSE NULL
  baldate = ndx.rec<1, datepos>
  IF baldate THEN
    acdate = account:*':baldate
    READV balance FROM accbals, acdate, 1 THEN
      CRT 'Accont balance was: ':0CONV(balance, 'MR2,$'
    END ELSE
      CRT 'Balance not found'
    END
  END ELSE
    CRT 'Index file is corrupt'
  END
END
REPEAT
```

This requires two reads to determine the account balance – regardless of the date requested. The first read gets the index item, while the second read gets the account balance.

Let's run through the code:

- We enter a date – let's say it is 14 December, 2009. This is internal date 15324. In practice, we would need to validate this date to ensure it is not in the future
- We enter an account – let's say it is: 123456. In practice, this would need some further validation
- We generate an ID for the index file. This is: 123456*200912
- We read the index item. This is as shown on the previous page
- We LOCATE the entered date in the index item. This returns position 1 because this is the position that we would need to insert the entered date (15324) to maintain the sequence in descending order

- We generate the ID for the ACCBALS file based on the account number and the date in the first position in the index file. This would be: 123456*15317
- We read and display the balance from the ACCBALS file.

Let's say the date entered was the 2nd of December. This has an internal value of 15312.

If we LOCATE this date in our index item, the position returned is 3. The date at position 3 is 15310 which is November 30 – the previous month-end value.

Note that in this usage of LOCATE, it doesn't actually matter whether we find the value in the list or not. Therefore, we simply set the ELSE action to NULL.

This banking application example is a typical usage of LOCATE. Of course, this only shows one part of the usage within the application – there will be an equivalent usage of LOCATE during the maintenance of the index item so that the index item is always in sorted order.

FIND

FIND is closely related to LOCATE. It finds a data element within a dynamic array. However, it does so in a different manner than LOCATE:

- LOCATE searches within one dimension of a dynamic array, and returns a position for the search string whether or not it is found. THEN and/or ELSE statements are executed depending on whether the string was found.
- FIND searches in one, two, or three dimensions – depending on the number of variables specified – and returns the position of the string in that number of dimensions. THEN and/or ELSE statements are executed depending on whether the string is found.

To make things a little clearer: FIND returns the position of the string in 1, 2, or 3 dimensions. In contrast, LOCATE only ever returns the position of the string within a single dimension of the dynamic array.

The syntax for FIND is:

```
FIND string IN dyn.array {,occurrence) SETTING field {,value {,subvalue}} {THEN statements } {ELSE statements}
```

As with LOCATE, the search string should make up the entire field, value, or subvalue. Unlike LOCATE, the values returned do not represent the position in the dynamic array where the string should be inserted – they only represent the string's actual position.

Warning: If the string is not found, then the values of field, value, and subvalue are not changed. Therefore, you cannot assume that the presence of data in these variables indicates that the string has been found – unless the variables were empty before the FIND statement was executed.

Program control structures

SDBasic supports variants on the common program control structures. In many cases, the SDBasic variants are more flexible than those found in languages such as *Visual Basic*, or *C/C++*.

IF-THEN-ELSE

SD supports both single and multi-line versions of IF statements:

```
IF condition {THEN statement} {ELSE statement}
IF condition {THEN
  statements
END} {ELSE
  statements
END}
```

In either construct, at least one of the THEN or ELSE branches must exist. This means that you can write statements such as:

```
IF ok ELSE CONTINUE
```

While this is a perfectly valid statement, it is difficult to read. Most people logically expect a THEN condition as the primary branch of an IF statement, and would find the following statement easier to read:

```
IF NOT(ok) THEN CONTINUE
```

Multi-line IF statements must have an END terminating each block of conditional statements:

```
IF condition THEN
  statements
END
```

The conditions shown in the examples use Boolean values. *SD* takes any non-null, non-zero value to be true, and null or zero to be false. Therefore, if the variable *ok* shown in the example above contained the value 'Y', then this would be true¹⁶, whereas if it contained an empty string, then this would be false. **Warning:** If the variable contained 'N', then this would also evaluate to true.

Boolean tests are good for evaluating the presence or absence of data. They should not be used for testing between two different data values. In that case, use a construct like:

```
IF UPCASE(answer) = 'Y' THEN
  statements
END ELSE
  statements
END
```

Alternatively, you could convert the original data to a Boolean value:

```
ok = (UPCASE(answer) EQ 'Y')
IF ok THEN ...
```

¹⁶ Note that some multi-value databases require Boolean values to be strictly numeric or null. Therefore, a value of 'Y' will result in a non-numeric value error, with zero assumed – i.e. false.

CASE

CASE statements provide an alternative method of branching for conditional execution. While IF-THEN-ELSE statements provide a two-way branch (or more if conditions are nested), CASE statements allow multi-way branching. The basic format of the statement is:

```
BEGIN CASE
  CASE condition-1
    statements
  { CASE condition-2
    statements }
  { CASE condition-n
    statements }
  { CASE 1
    statements }
END CASE
```

The statement begins with the BEGIN CASE declaration, and ends with the END CASE declaration. In between these two declarations, there are as many CASE conditions as required.

In operation, code execution in a CASE statement always takes the first condition that evaluates to TRUE. Once that conditional block of statements has been executed, all the remaining cases are skipped and execution continues with the statement immediately following the END CASE statement.

Optionally, you can include a CASE 1 condition. This condition always evaluates as TRUE and therefore acts as a default execution branch (or ELSE clause).

Note the following points about CASE statements:

- The conditions do not need to be related to each other – although they often are. This is in contrast to the SELECT CASE statements in some other languages that can only branch on the value of a single variable
- If no condition is matched, then execution will continue with the first statement following END CASE

Consider the following code fragment:

```
cntCrLf = COUNT(descdata, CR:LF)
cntCr = COUNT(descdata, CR)
cntLf = COUNT(descdata, LF)

BEGIN CASE
  CASE cntCrLf EQ cntCr AND cntCrLf EQ cntLf; fdelim = CR:LF
  CASE cntCr GT cntLf; fdelim = LF
  CASE cntLf GT cntCr; fdelim = CR
  CASE 1; fdelim = ''
END CASE
```

This code counts the number of carriage returns and line feeds in the variable *descdata*, then assigns the variable *fdelim* on the basis of these counts. Although a CASE 1 condition is present, this should be impossible to reach.

Loops

Conditional loops

The basic structure of a conditional loop is:

```
LOOP
  {statements}
  {WHILE | UNTIL condition {DO}}
  {statements}
REPEAT
```

The `DO` keyword is not required in *SD*, but is required by other multi-value databases, and so is shown here for compatibility.

The above structure shows two blocks of optional statements. The first set of optional statements will always be executed within the loop. However, the second set will only be executed when the condition statement allows.

A typical usage of this structure is in the sequential processing of records in a file:

```
EQUATE CUST.SURNAME TO 4
EQUATE CUST.FIRSTNAME TO 5
OPEN 'CUSTOMERS' TO customers ELSE STOP 201, 'Customers'

eof = @FALSE
SELECT customers
LOOP
  READNEXT custid ELSE eof = @TRUE
UNTIL eof DO
  READ custrec FROM customers, custid THEN
    CRT custrec<CUST.FIRSTNAME>: ' ': custrec<CUST.SURNAME>
  END
REPEAT
```

The assignment of the *CUST.FIRSTNAME* and *CUST.SURNAME* references, and the opening of the *CUSTOMERS* file are shown for completeness.

This program fragment works as follows:

- The `SELECT` statement selects the *customers* file and returns a select list for processing
- The `READNEXT` statement reads the next customer id from the select list. If there are no more items in the select list then the *eof* variable is set to `@TRUE`
- The `UNTIL` clause tests the value of the *eof* variable. If *eof* is `@FALSE`, then processing moves to the next part of the loop. If *eof* is `@TRUE`, then loop processing ends, and processing continues with the statement following `REPEAT`
- The inner part of the loop reads the customer record, and displays the customer's firstname and surname
- When the `REPEAT` statement is encountered, processing returns to the `LOOP` statement, and the next customer id is read from the select list.

Loops can also be written with only one internal block of statements:

```

ii = 0
LOOP
  ii += 1
  more statements
UNTIL ii GE maxii DO REPEAT

ok = @TRUE
LOOP WHILE ok DO
  more statements
REPEAT

ok = @TRUE
LOOP
  more statements
WHILE ok DO REPEAT

```

Note that the WHILE/UNTIL conditions may occur at any point during the loop – unlike some other languages that require WHILE conditions to be placed at the top of the loop, and UNTIL conditions at the end. Similarly, you may have more than one WHILE/UNTIL conditions within a loop – although such structures may be difficult for others to read.

They can also be written without any WHILE/UNTIL conditions:

```

SELECT customers
LOOP
  READNEXT custid ELSE EXIT
  READ custrec FROM customers,custid THEN
    CRT custrec<CUST.FIRSTNAME>:' ':custrec<CUST.SURNAME>
  END
REPEAT

```

In this case, the EXIT statement is used to terminate the loop when the list of customer ids has been exhausted. More information on the EXIT statement will be given shortly.

For-Next loops

If you know the number of loops that you require, you can use a FOR-NEXT loop rather than testing a condition on every loop. The basic structure of FOR-NEXT loops in *SD* is similar to that in other languages, but does have some variations:

```

FOR var = start TO end { STEP stepsize }
  statements
NEXT var

```

If STEP is omitted, then ‘stepsize’ is assumed to be 1.

For example:

```

IF st GT '' THEN
  dc = DCOUNT(st, @AM)
  FOR ii = 1 TO dc
    CRT msgs<ii>
    EXECUTE st<ii> CAPTURING junk
  NEXT ii
END

```

In this example, the variable *st* may contain a series of executable statements. First, the variable is tested to see if it contains any data. If so, the statements are counted, and a loop started which executes each statement and displays a message associated with each statement.

SD also allows `WHILE` and `UNTIL` statements to be included in the `FOR-NEXT` loop. Consider the following program:

```
PROGRAM TEST
kk = 0
FOR ii = 1 TO 4
  FOR jj = 1 TO 6
    kk = ii * jj
    WHILE kk LE 12
      CRT kk 'R#6':
    NEXT jj
  NEXT ii
  CRT
NEXT ii
STOP
END
```

The output from this program is:

```
RUN BP TEST
  1      2      3      4      5      6
  2      4      6      8     10     12
  3      6      9      12
  4      8     12
```

In this case, the inclusion of the `WHILE` clause has made the display of the number conditional on its value.

Recent versions of *SD* have two new syntax options for the `FOR-NEXT` loop:

```
FOR var = value1 {,value2 ...}
  statements
NEXT var
```

and:

```
FOR EACH var IN string
  statements
NEXT var
```

The first of these two new variants allows you to specify a list of values to use in the `FOR-NEXT` loop. The second allows you to use the values contained in a dynamic array within the loop. See the online help for more information on these two variants.

Some programs may be written to use the final value of the index variable after the loop has terminated. For example:

```
PROGRAM TEST
FOR ii = 1 TO 5
  CRT ii
NEXT ii
CRT 'Final value is ':ii
END
```

This produces the output:

```

RUN BP TEST
1
2
3
4
Final value is 4

```

Some other multi-value databases may display 5 as the final value. *SD* can mimic this behaviour by using the compiler mode `FOR.STORE.BEFORE.TEST`. This could be included directly in the program or stored in a `$BASIC.OPTIONS` item either in the program file or in the account VOC:

```

PROGRAM TEST
$MODE FOR.STORE.BEFORE.TEST
FOR ii = 1 TO 5
  CRT ii
NEXT ii
CRT 'Final value is ':ii
END

RUN BP TEST
1
2
3
4
Final value is 5

```

While it is possible to make *SD* emulate this behaviour, you should really question whether this is sensible. The index value of a `FOR-NEXT` loop only has meaning within the loop. You should not rely on its value after loop termination. It isn't difficult to write your code in such a manner that you don't have to worry about this difference in behaviour between multi-value systems.

EXIT and CONTINUE

`EXIT` and `CONTINUE` statements may be used to modify the behaviour of both conditional loops and `FOR-NEXT` loops.

- `EXIT` causes the loop to terminate.
- `CONTINUE` skips the remaining statements in this loop and starts a new loop

Example usage of both `EXIT` and `CONTINUE` is shown below:

```

SELECT customers
LOOP
  READNEXT custid ELSE EXIT
  READ custrec FROM customers,custid THEN
    IF custrec<CU.ACTIVE> NE 'Y' THEN CONTINUE
    GOSUB processcust
  END
REPEAT

```

In this loop, if there are no more customer id's to process, then the `EXIT` statement will cause the loop to terminate. Valid customers are checked to see if they active. If they are not active, then the `CONTINUE` statement causes processing to skip the processing of the customer record and jump to the `REPEAT` statement.

Subroutines

Subroutines are a means of breaking your program into small blocks that:

- allow you to structure to your program
- encourage re-use of code sections.

The key concepts of a subroutine are:

- it is a block of code that carries out a specific action or set of actions
- it is called from elsewhere in the program via the GOSUB statement for internal subroutines or the CALL statement for an external subroutine
- once processing of the subroutine is complete, control returns to the statement immediately following the calling statement.

Subroutines may call other subroutines, which in turn may call other subroutines. There are limits to the depth of such nesting (the default setting is 1,000 subroutine calls), but you are unlikely to reach them unless an error of program logic causes recursive or circular subroutine calls.

Program structure

An example of structured programming is shown in the loop above. What we see is a relatively small block of code, the purpose of which is readily apparent.

In contrast, consider what would happen if we included the customer processing code inside the loop. Say the code for the customer processing code was 200 lines long. In that case, we would not be able to quickly see the start and end points of the loop, and we would probably lose track of flow of logic.

Well structured programs usually have logic flows that look like:

```
GOSUB initialise
GOSUB getuserresponsesmsg); * Set colours
END ELSE
  msg = 'Error - Foreground colour same as background colour'
END

IF msg GT '' THEN;          * Errors encountered
  CRT msg;                  * Display error message
  GOSUB showusage
  STOP
END
*
RETURN
```

The above code fragment calls the internal subroutine *getcolournum* twice, and the external subroutine *SY.SET.COLOUR* once. Both of these subroutines are also called from elsewhere in the program, and the *getcolournum* subroutine itself calls another external subroutine:

```

getcolournum:
*
CALL SY.GET.AT.COLOUR.NUM(thiscolour, colournum, msg)

IF msg GT '' THEN;          * Colour not found - error message
  CRT msg
  GOSUB showusage
  INPUT pause,1
  STOP
END
*
RETURN

```

Internal subroutines

Internal subroutines are defined within the main body of the program. They start with a label that identifies the subroutine, and end with a RETURN statement. Labels are normally¹⁷ an alphanumeric string followed by a colon (:), or a number (which may optionally be followed by a colon). For example:

```

initialise:
  statements
RETURN

```

OR:

```

1000
  statements
RETURN

```

You can put a comment on the same line to identify the purpose of the subroutine if it is not immediately apparent:

```

1000 ;* Initialise variables
  statements
RETURN

```

The above subroutines would be called with the following statements:

```
GOSUB initialise
```

```
GOSUB 1000
```

External subroutines

External subroutines are stored outside the main program – they are program modules in their own right. This means that external subroutines may be called by any program, whereas an internal subroutine can only be called from within its parent program.

External subroutines have another distinguishing characteristic – they can be defined to take a list of parameters that define their action. In contrast, internal subroutines make use of the same set of variables that are used by the parent program.

An external subroutine begins with the SUBROUTINE declaration, and ends with a RETURN statement.

```

SUBROUTINE subname{(parameter {,parameter ...})}
  statements
RETURN

```

¹⁷ A third label format is also available. See the online help for more information.

For example:

```
SUBROUTINE SY.GET.SETTING(ctrlldata, identifier, settings, found)
* ----- *
*
* Copyright 2008 Rush Flat Software
*
* Version: 1.0.0
* Author : BSS
* Created: 20 Mar 2006
* Updated: 20 Mar 2006
*
* Subroutine to search the passed control data for a particular
* identifier. Subroutine passes back the settings and found
* variables.
*
* Assumes that the control data is in the form:
*
* identifier1=settings1
* identifier2=settings2
* etc
* ----- *
*
$CATALOGUE GLOBAL

progname = 'SY.GET.SETTING'

upctrlldata = OCONV(ctrlldata, 'MCU')
upidentifier = OCONV(identifier, 'MCU')
numctrllines = DCOUNT(ctrlldata, @AM)

found = @FALSE
settings = ''
IF numctrllines GT 0 THEN
  ii = 0
  LOOP
    ii += 1
  UNTIL ii GT numctrllines OR found DO
    thisline = ctrlldata<ii>
    upthisline = OCONV(thisline, 'MCU')
    temp = TRIM(FIELD(upthisline, '=', 1))
    IF temp = upidentifier THEN
      settings = TRIM(FIELD(thisline, '=', 2))
      found = @TRUE
    END
  REPEAT
END
*
*
RETURN
* ----- *
*
END
```

And this subroutine would be called as follows:

```
identifier = 'colours'
CALL SY.GET.SETTING(sysctrl.userdata, identifier, colours, found)
colours = OCONV(colours, 'MCU')
```

So, if the control data that is passed to the subroutine looks like:

```
Colours=darkblue,yellow
NormCols=132
NormRows=35
ExtCols=160
ExtRows=40
ScrMode=Normal
```

then, the above call would return the string 'darkblue,yellow' in the variable *colours*, and @TRUE in the variable 'found'.

Let's go through the subroutine and call in a bit more detail:

- The subroutine was declared with the SUBROUTINE statement, and the declaration contained four variables to be passed to or from the subroutine.
- The subroutine contained a short description of what it does, followed by the actual code. This description could have been more explicit in the values that need to be passed to the subroutine, and those that will be returned.
- This code ended with a RETURN statement.
- The call to the subroutine passed four variables as part of the call. Note that the names of these variables do not have to match the names declared in the subroutine code. Inside the subroutine, the passed variables take on the names declared in the subroutine heading. Therefore, the variable *sysctrl.userdata* that is passed to the subroutine is referred to as *ctrldata* inside the subroutine.
- While the description implies that you should pass *ctrldata* and *identifier*, and the subroutine will pass back *settings* and *found*, this is a human interpretation of what happens. The subroutine itself makes no distinction between the variables in the declaration. *If a variable declared in the subroutine header is changed in the subroutine, it will be passed back in its changed state.* Therefore, if the subroutine changed the variable *identifier* to the literal value 'junk', then the value 'junk' would be available to the calling program when control returns there.

This last point is quite important. You need to be clear about the way subroutines change the variables passed to them. There are various strategies for ensuring that some variables are not changed by the subroutine:

- Pass a copy of the variable to the subroutine:

```
temp = sysctrl.userdata
CALL SY.GET.SETTING(temp, identifier, colours, found)
```

- Pass the variable by value. This ensures that only the value of the variable is passed to the subroutine – not the variable itself. To pass the variable by value, enclose the variable in parentheses in the the call:

```
CALL SY.GET.SETTING((sysctrl.userdata), identifier, colours, found)
```

- Define the subroutine parameters as being passed by value. This ensures that the variable passed to the subroutine will not be changed. To declare the variable as being passed by value, enclose the variable in parentheses in the declaration:

```
SUBROUTINE SY.GET.SETTING((ctrlldata),identifier,settings,found)
```

Which approach you use depends on what you expect the subroutine to do. Often, the point of the subroutine is to change the value of the variable, in which case you don't need the strategies outlined above.

Local subroutines

SD has a third type of subroutine which has features of both internal and external subroutines:

- The subroutine is defined within the main body of the program
- The subroutine can be defined to take parameters

These subroutines are defined using the LOCAL keyword, and have an END statement after the RETURN:

```
LOCAL SUBROUTINE subname{(parameter {,parameter ...})}
  statements
RETURN
END
```

Essentially, the RETURN statement terminates the SUBROUTINE declaration, while the END is required to terminate the LOCAL declaration.

Local subroutines may also employ local variables – but these must be explicitly declared using the PRIVATE keyword. Otherwise, variables in local subroutines are global in scope.

Local subroutines are called using the GOSUB statement.

```
PROGRAM TEST
dt = ''
st = '28/2/10'
GOSUB DATATYPE(st, dt)
CRT 'Datatype of ':st:' is ':dt

st = '123.45'
GOSUB DATATYPE(st, dt)
CRT 'Datatype of ':st:' is ':dt
CRT temp
STOP

LOCAL SUBROUTINE DATATYPE(datastring, datatype)
PRIVATE temp, datetest
temp = datastring
CONVERT ',' TO ' ' IN temp
datetest = OCONV(ICONV(temp, 'D'), 'D2/')
IF datetest NE '' AND LEN(temp) < 6 THEN datetest = ''
BEGIN CASE
CASE temp = ''; * NULL is Text
datatype = 'T'
CASE INDEX(temp, ' ',1); * At least one space
datatype = 'T'
CASE OCONV(temp, 'MCA') NE ''; * Alpha is not null
datatype = 'T'
CASE NUM(temp); * Is numeric
datatype = 'N'
CASE datetest NE ''; * OCONV DATE is not null
datatype = 'D'
CASE 1; * Anything else = text
datatype = 'T'
END CASE
RETURN
END
END
```

BASIC BP TEST

```
Compiling BP TEST
***
WARNING: TEMP is not assigned a value
0 error(s)
Compiled 1 program(s) with no errors
```

RUN BP TEST

```
Datatype of 28/2/10 is D
Datatype of 123.45 is N
00000109: Unassigned variable TEMP at line 10 of D:\SD\SDINTRO\BP.OUT\TEST
```

The above example shows how to define and call a local subroutine. It also shows how the PRIVATE variables contained within the local subroutine are not available to the main program.

User Defined Functions

User defined functions are broadly similar to external subroutines (and like subroutines, can also be defined as LOCAL). However, unlike subroutines, they use a general assignment syntax rather than a call:

```
result = MYFUNCTION(arg-list)
```

Like external subroutines, functions usually occupy their own operating system file, and have the general form:

```
FUNCTION functionname{(parameter {,parameter ...}) {VAR.ARGS}}
statements
RETURN varname
```

For example:

```
FUNCTION SY.EXCELDATE(datestring)
*
* Version: 1.0.0
* Author : BSS
* Created: 08 Mar 2007
* Updated: 08 Mar 2007
*
* Copyright 2008 Rush Flat Software
*
* ----- *
*
$CATALOGUE GLOBAL

    internaldate = ICONV(datestring, 'D')
    xldate = internaldate + 24837
RETURN xldate
*
END
```

This function takes a passed date in external format (e.g. 25 Apr 2009), and returns this as an Excel date number.

To use the function in a program, we must define it so that the compiler knows this is a valid function and can validate the number of arguments passed at compile time. We use the DEFFUN keyword to define the function in a program:

```
PROGRAM TEST
DEFFUN SY.EXCELDATE(datestring)
testdate = '25 Apr 2009'
CRT SY.EXCELDATE(testdate)
END
```

And the output is:

```
RUN BP TEST  
39928
```

Note that we have been able to call the function directly in the CRT statement, although a more normal usage of this function may have been:

```
xdate = SY.EXCELDATE(testdate)
```

As noted above, *SD* allows functions to be defined locally within a program through use of the LOCAL statement. See the subroutines section above for more information on this, or lookup LOCAL in the online help.

The other twist provided by *SD* is to allow a variable number of arguments through use of the VAR.ARGS keyword in the function definition:

```
FUNCTION FNTEST(arg1, arg2, arg3, arg4) VAR.ARGS  
  x = ARG.COUNT()  
  RETURN x  
END
```

This function has been defined to accept up to 4 arguments. However, all it does is return the count of arguments passed to it. The following program shows this function in use:

```
PROGRAM TEST  
DEFFUN FNTEST(arg1, arg2, arg3, arg4) VAR.ARGS  
numargs = FNTEST('ABC')  
CRT 'Number of arguments = ':numargs  
END
```

```
RUN BP TEST  
Number of arguments = 1
```

Note that if you wish to use functions with a variable number of arguments, the VAR.ARGS keyword should be included in both the function definition and the DEFFUN declaration – although if you test this, you can omit it from the function definition, but it must be included in the DEFFUN declaration.

As with subroutines, variables are passed to the function by reference. This means that changes to the variables in the function will be carried back to the calling program. Note this is different from many other computer languages which pass arguments to functions by value. You can use any of the strategies outlined in the section on subroutines to pass the arguments as values.

Files

SD applications use files extensively. Therefore, it is vital to understand how to use files in *SD*.

In general, the processes involved in dealing with files are:

- opening the files
- selecting records in the files
- reading the records from the files
- writing records to the files

- closing files.

There are several related issues to consider also:

- error handling
- record locking
- handling special file types

The following sections will give a brief coverage of these issues.

Most file handling statements have an {ON ERROR statements} clause within them. These statements are executed when a serious error condition is encountered in the file structure. As this clause is common to most statements, it will be omitted in the following descriptions.

Likewise, most file handling statements have optional THEN and ELSE clauses. While these are noted as being optional, in reality, they must have at least one of these clauses present.

In all cases, see the online help for more information.

Opening files

Files must be opened before they are available within an SD program. Opening the file associates the file's operating system filename with a variable within the program:

```
OPEN filename TO filevar {THEN statements } { ELSE statements }
```

The filename may contain a reference to a dictionary so that you can open the dictionary itself, or the data portion of a multifile:

```
OPEN 'CUSTOMERS' TO customers ELSE GOTO fileopenerror
```

```
OPEN 'DICT','SALES' TO sales.dict ELSE GOTO fileopenerror
```

```
OPEN 'SALES','FY2009' TO sales ELSE GOTO fileopenerror
```

The dissociation of the database filename with the internal file variable means that you could use the same file variable for multiple files (one at a time of course). In the multifile example given above, we may have the year (say 2008) we wish to open in the variable *fyear*. Therefore:

```
fname = 'FY':fyear  
OPEN 'SALES',fname TO sales ELSE GOTO fileopenerror
```

This is a particularly valuable technique for two reasons:

- It allows you to write generalised software that operates on a number of files. The physical file reference that you provide to the software is converted to a file variable for the actual operations.
- The error handling can be generalised. Once again, pass the error handler the name of the file in a variable, and it can output an appropriate message.

The file variable is just another variable that should conform to *SD* naming rules. However, as file variables have specific roles within *SD*, you should try to be consistent with your naming of file variables. Some strategies are:

- Use the file name as the name of the file variable:

```
OPEN 'STAFF' TO staff ELSE ...
```

- Use a file variable that indicates purpose and source:

```
OPEN 'CUSTOMERS' TO cust.file ELSE ...
OPEN 'DICT','REPORTS' TO reports.dict ELSE ...
```

- A common variant on this is to use '.f' as a suffix (or 'f.' as a prefix) to denote a file variable:

```
OPEN 'CUSTOMERS' TO customers.f ELSE ...
```

Consistency of naming file variables will aid subsequent programming enormously, as you don't have to continually check what name was given to each file.

The `ON ERROR` clause will only get executed if severe errors are encountered when opening the file. A `THEN` clause will be executed if the file is opened successfully, while an `ELSE` clause will be executed if the file cannot be opened. At least one of the `THEN` or `ELSE` clauses must be present.

Error handling

A common form of error handling on `OPEN` statements is simply to stop the program with an error message. This is fine in simple applications, but is not appropriate in larger applications where the `OPEN` error may occur deep in the application.

A typical example of this type of error handling is:

```
OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
```

201 refers to the error message number in the `ERRMSG` file. If the `OPEN` statement fails, then this produces the following error message:

[201] 'Sales' IS NOT A FILE NAME

Importantly, the `STOP` statement actually stops the program, so this is a drastic form of error handling. Ideally, we want to open files in a way that captures errors and gives a chance to handle them appropriately. One way to do this is to wrap the standard `OPEN` statement in a custom subroutine (or function) that allows you set the desired action when you call the subroutine, and it passes back appropriate messages:

```

SUBROUTINE FILE.OPEN(filename, fileptr, errorlevel, etext)
*****
* Bp File.Open - A subroutine to open files in a standardised manner.
*
* Copyright 2008 Rush Flat Software
*
* Author : BSS
* Created: 09 Aug 2008
* Updated: 06 Aug 2009
* Version: 1.0.1
*
* Pass   : filename
* Return : fileptr, errorlevel, etext
*
* Errorlevel: 0 - No errors
*             1 - Errors encountered
*             2 - Severe error encountered
*
* ----- *
*
$CATALOGUE GLOBAL

errorlevel = 0
etext = ''
fileopened = @FALSE

CONVERT ' ' TO '' IN filename
ofilename = filename

dictname = ''
IF INDEX(filename, ',', 1) THEN
    dictname = FIELD(filename, ',', 1)
    filename = FIELD(filename, ',', 2)
END

fileptr = ''
BEGIN CASE
    CASE (dictname EQ '') OR (dictname = filename)
        GOSUB opendata
    CASE dictname EQ 'DICT'
        GOSUB opendict
    CASE 1
        filename = dictname:',':filename
        GOSUB opendata
END CASE

RETURN
STOP
*
* ----- *
*
opendata:
*
OPEN filename TO fileptr ON ERROR
    GOSUB openerror
    errorlevel = 2
END THEN
    fileopened = @TRUE
END ELSE
    GOSUB openerror
END

RETURN
*
* ----- *
*
opendict:
*
OPEN 'DICT',filename TO fileptr ON ERROR
    GOSUB openerror
    errorlevel = 2
END THEN
    fileopened = @TRUE
END ELSE
    GOSUB openerror
END

RETURN

```

```

*
* ----- *
*
*
* openerror:
*
* errorlevel = 1
* errcode = STATUS()
* CALL !ERRTEXT(etext, errcode)
* etext = 'Err ':errcode': Error opening file: ':ofilename
*
* RETURN
*
* ----- *
*
* END

```

The *FILE.OPEN* subroutine gets the error description using the !ERRTEXT standard subroutine that is supplied with *SD*. This subroutine converts error numbers (supplied by the STATUS() function) to descriptive text.

Selecting data in files

Selecting data from a file is generally understood to mean selecting a set of the available records from a given file. In the multi-value world, it has the added implication that the ID's of the selected records will be available in a list (a select-list).

There are two basic ways of creating a select-list – using an internal select, and using an external select. An internal select is carried out within the BASIC program, while an external select is executed outside the program.

Once the program has a select-list, it will usually loop through the ID's in the list, and process the associated records.

Internal select

An internal select has the syntax:

```
SELECT var {TO list-num}
```

The behaviour of this statement can be altered via a \$MODE compiler directive to select to a list variable¹⁸ rather than a list number. *SD* also has variants of the SELECT statement that always select to a list number (SELECTN) or a list variable (SELECTV).

18 The choice of whether to use list numbers or list variables is usually made on the basis on the developers background. If they have come from a PICK background, they will usually choose list variables, while those from an Information background will choose list numbers. They are functionally similar but have slight differences in usage.

At this stage, the option to select to a list number or list variable will be ignored. This only needs to occur if multiple select lists could be concurrently active. However, it is good practice to always select to a list number/variable for two reasons:

- multiple select lists can be more easily handled when they occur; and
- to ensure that no active select lists are left behind by your programs.

This second point needs a bit more explanation. Consider the case where a program generates a select-list, and does not associate it with a list number or variable. This means that it will be the default select list. The program then begins processing the list, but stops before all items in the list are exhausted. The remaining items in the list will remain active, and will be processed by any subsequent READNEXT command. Further, the list can even continue to exist after the program terminates, and will be processed by any subsequent SDQuery commands or programs.

In short, a default select-list that has not been exhausted can cause programs or SDQuery commands to behave in an unexpected fashion. To avoid the problems described above, always select to a list number or variable.

The *var* that is selected by SELECT may be either the file variable of an open file, or a variable containing a field-mark delimited list of record ID's. In either case, an internal select simply selects all records referenced by *var* into a select-list. The statement has no ability to select a subset of the records, nor to sort them into any order. The order of the record ID's in the list simply reflect the order of the records in the file or variable.

SD has another variant of the SELECT statement which does a simple sort of the record ID's (SSELECT). However, as this sort is limited to a left-justified ascending sort order, this is of limited use.

The advantage of an internal select is that it is fast.

External select

An external select uses the SDQuery selection commands to select some or all of the record ID's in a given file. The command may also sort the records into a specific order.

SDQuery commands were covered in Part 1 of *Getting Started in SD*. This document will only cover the means by which these commands are used within SDBasic.

The issue for SDBasic is how to run a SDQuery command (or any other command that is normally run from the command line). This is done using the EXECUTE or PERFORM statements. EXECUTE has a number of optional clauses which are not covered here – see the online help for more information.

```
EXECUTE command {CAPTURING display}
```

For example:

```
EXECUTE \SSELECT IRATES WITH YEAR EQ "2006" \ CAPTURING junk
```

This would return a select-list of record ID's in the IRATES file where the year was 2006. It uses the SSELECT command without specifying a sort order, so the record ID's would be sorted into their ID order

– which in this case is ascending month order within the year.

If we run this command from the command prompt, we get:

```
:SSELECT IRATES WITH YEAR EQ "2006"  
12 record(s) selected to list 0  
::
```

The ‘CAPTURING junk’ part of the EXECUTE statement captures the message that is reported by SDQuery. This stops the message from being displayed, and upsetting any screen formatting that you have.

Note that the command that is executed must be passed as either a quoted string, or as a variable. In the above example, the backslash character (\) has been used to quote the string. The backslash is useful for this purpose as it allows both single and double quotes to be used within the SDQuery command.

The same command could be executed using a variable as follows:

```
cmd = \SSELECT IRATES WITH YEAR EQ "2006"\  
EXECUTE cmd CAPTURING junk
```

Any of the SDQuery selection commands can be used in this manner. The usual ones are SELECT, SSELECT, and QSELECT. However, the stored list commands can also be used – e.g. GET-LIST.

The advantages of an external select are that it allows you to select a subset of records, and return the list in a sorted order.

Which selection should I use?

In general:

- If you want to select most or all records in a file and the order of the records is not important, then use an internal select
- If you want to select a small subset of records and/or you want the record ID’s in a specific order, then use an external select.

Reading from files

Once you have a record ID, you can READ the associated record from the file:

```
READ var FROM filevar, record-id {THEN statements} {ELSE statements}
```

For example:

```
READ sales.vec FROM sales.summary, idate ELSE  
CRT 'Sales data for ':OCONV(idate, 'D'):' not on file'  
END
```

This looks up data from the sales summary file which has a key of the internal date number. If the record is not found on the file, then the ELSE clause is executed which displays an error message.

The file must have been opened before the READ statement is attempted. Note also that the READ statement accesses the file variable – not the physical file name.

There are a number of READ statements:

- READV reads a single field from the record rather than the whole record. The syntax for this statement requires that the field number be included.
- READL and READU read the whole record and place a lock on the record to prevent other users from updating it. Record locking will be covered later under ‘multi-user issues’.

It is important to recognise that the READ statement will ALWAYS read the record – regardless of the state of any record locks. If your application is only reading the record to obtain a value from the file, then READ is appropriate. However, if the application is going to update the record, then you should use the READU statement which applies an update lock to the record as part of the read process. This will be covered in more detail later.

Getting the ID from the select list for the READ

The above two sections cover creating a select-list of ID’s, and then using an ID to read from the file. However, we need an intermediate step to get an ID from the select-list for use in the READ statement.

The READNEXT statement takes an ID from a select-list and stores it in a variable:

```
READNEXT var {FROM list} {THEN statements} {ELSE statements}
```

For example:

```
EQUATE CUST.SURNAME TO 4
EQUATE CUST.FIRSTNAME TO 5
OPEN 'CUSTOMERS' TO customers ELSE STOP 201, 'Customers'

eof = @FALSE
SELECT customers
LOOP
  READNEXT custid ELSE eof = @TRUE
UNTIL eof DO
  READ custrec FROM customers, custid THEN
    CRT custrec<CUST.FIRSTNAME>: ' ':custrec<CUST.SURNAME>
  END
REPEAT
```

The READNEXT statement in the unconditional portion of the LOOP gets an ID from the select-list, and assigns it to the variable *custid*. If there are no more ID’s in the list, then the statement assigns a value of @TRUE to the variable *eof*.

Note that you can only read FORWARD through a select-list. You cannot back up through a select-list. (There is no READPREV statement to get the previous item-id).

An alternative way of processing the select-list is to use READLIST statement to read the entire list into a variable, and then process the variable:

```
SELECT customers
READLIST custlist THEN
  LOOP
    REMOVE custid FROM custlist SETTING delim
    READ custrec FROM customers,custid THEN
      CRT custrec<CUST.FIRSTNAME>:' ':custrec<CUST.SURNAME>
    END
  WHILE delim DO REPEAT
END
```

READLIST reads the entire select-list into a variable, while the REMOVE statement extracts the next part of the dynamic array. See the documentation for more information.

Writing to files

Writing a record to a file uses the WRITE statement:

```
WRITE var TO filevar, record-id
```

The keyword ON may be used instead of TO.

For example:

```
WRITE sales.vec ON sales.summary, idate
```

Note the WRITE statement has no THEN or ELSE clauses.

If the record-id already exists in the file, then the existing record will be overwritten. Indeed, the WRITE statement offers no way for the programmer to determine whether an item already exists. Any such management of existing items must be done using the THEN or ELSE clauses of the READ statement.

There are also WRITE statements that match the variants of the READ statements:

- WRITEV writes a single field to the record rather than the entire record.
- WRITEU writes the whole record and maintains the record lock.

All of these WRITE statements will write the item regardless of the state of any record locks. See the online help, or the 'Multi-user issues' section later in this document for more information.

Closing files

Multi-value systems do not usually need to explicitly close files. Nevertheless, there is a CLOSE statement which will close the file – or remove the association between the physical file and the file variable:

```
CLOSE filevar
```

Other methods of file handling

The file handling so far has dealt with files defined within the local account, or which have a Q-pointer in the voc pointing to the file in another account. In these cases, *SD* uses the data stored in the voc entry to find the location of the file in the file system.

But what if the file isn't defined in the voc? Then you can supply the path to the file directly:

```
OPENPATH pathname TO filevar {THEN statements} {ELSE statements}
```

This is similar to the `OPEN` statement¹⁹ except that you are supplying a pathname rather than an *SD* filename. This method can be used to open both dynamic files and directory files in other *SD* accounts. It can also be used to open other folders in the file system:

```
OPENPATH 'C:\Temp' TO temp.folder THEN ...
```

You can also quickly read a single item in an operating system file (rather than opening the file and then reading the item):

```
OSREAD var FROM path {THEN statements} {ELSE statements}
```

For example:

```
OSREAD txt FROM 'C:\TEMP\TEST.TXT' ELSE txt = ''
```

There is a corresponding `OSWRITE` statement to match `OSREAD`:

```
OSWRITE var TO path
```

Unlike the `WRITE` statement, you may not use the keyword `ON` instead of `TO`.

You can also read a text file sequentially – that is record by record or block by block by using the sequential file processing commands. These include:

```
OPENSEQ pathname {THEN statements} {ELSE statements}
READSEQ var FROM filevar {THEN statements} {ELSE statements}
WRITESEQ var TO filevar {THEN statements} {ELSE statements}
READBLK var FROM filevar, bytes {THEN statements} {ELSE statements}
WRITEBLK var TO filevar {THEN statements} {ELSE statements}
READCSV FROM filevar TO var1, var2 etc {THEN statements} {ELSE statements}
WRITECSV var1, var2 etc TO filevar {THEN statements} {ELSE statements}
CLOSESEQ filevar
```

See the online help for more information on these statements.

Multi-user issue

SD is a multi-user database system. As such, there will be times when multiple users try to access or update the same record at the same time. A good application will ensure that such contention issues are

¹⁹ Pathnames can also be used within the `OPEN` statement, but only if an extended syntax is “allowed”. See the `FILERULE` configuration parameter for more information.

handled in a standard manner that preserves data integrity while not inconveniencing users too much. This is achieved through record locking.

Before looking closer at the individual locking statements, it is important to understand the following points:

- Locking is maintained by the application – not by the database. If two different applications access the same file, then it is up to the developer to ensure that the locking is consistent between the applications
- Locking is (normally) advisory only – that is, the locking does NOT prevent applications from reading or writing to the file – UNLESS those applications have been written to respect the locks
- *SD* has a configuration parameter that changes this default behaviour. If the `MUSTLOCK` parameter is set to a value of 1, then any attempt to write or delete an item from a program where the program does not hold an update lock will result in a program abort. While use of this configuration parameter enforces better structure within programs, it will probably break many existing multi-value applications.

The practical implication of these points is that you should write all applications to use and respect locks. This way, when new applications are added to the system, you can be certain that all applications are handling locks correctly.

SD offers locking at two levels – whole file locking, and individual record locking. Given that locking the entire file has potential to inconvenience many users, this option should be used with care.

When should locking be used

Any program that updates data files – or may update data files – should use some form of locking. Even if the system is written as a single user application, it is still good practice to build in locking as (a) this will make it easy to convert it to a multi-user application; and (b) it will maintain a single style of coding between applications.

In some cases, you may also want to employ locking even when data is not being updated. For example, you may want to report on the state of the system at a particular instant in time. To be sure that the data is totally consistent for that instant, you may want to lock the entire file(s) for reporting.

Note that neither of these locking scenarios actually stops applications from reading or writing to the file – a `READ` will ALWAYS read from the file, and a `WRITE` will ALWAYS write to the file. However, if the application is written to test for locks (using `READU`), then the reads and writes will only occur in accordance with those locks.

File locks

To lock an entire file, use the `FILELOCK` statement:

```
FILELOCK filevar {LOCKED statements}{THEN statements}{ELSE statements}
```

The `LOCKED` clause will be executed if another user already has a file lock or a record lock on any record within the file. Unusually, the `THEN` and the `ELSE` clauses are completely optional, and neither need be present.

The lock should be released once processing of the file is complete. This is done using either the `FILEUNLOCK` or the `RELEASE` statements:

```
FILEUNLOCK filevar {LOCKED statements}{THEN statements}{ELSE statements}
RELEASE filevar
```

Note that this form of the `RELEASE` statement will release all locks associated with *filevar* – not just the file lock. See the online help for more information.

The usage of these statements will be something like:

```
OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
err = @FALSE
FILELOCK sales LOCKED err = @TRUE
IF NOT(err) THEN
  ...
  process file here
  ...
FILEUNLOCK sales
END
```

This example simply bypasses the file processing if a lock already exists on the file. In practice, error handling should be more sophisticated than this.

Record locks

SD supports two types of record locks – read (or shared) locks, and update locks. This section will mostly cover the use of update locks.

The purpose of an update locks is reasonably obvious – you place an update lock on a record when you want to update the record. The purpose of the read lock is a little less obvious – its basic action is to prevent an update lock from being applied. This allows an application to process the entire file without any risk that another user will update it during the processing. See the Locks section in the online help for more information on read locks.

The essential purpose of update locks is to allow the developer to structure the system so that multiple users read and write data in a consistent fashion. What we want to avoid is something like:

```
User A reads record
User B reads record
User B updates record
User A updates record
```

This will leave the system looking the way that User A expects, but User B's changes have been lost. The use of record locking would have allowed the above situation to have been trapped and action taken to avoid problems.

There are two basic approaches to the use of record locks. These approaches are sometimes termed optimistic and pessimistic locking:

Under pessimistic locking, the record is locked at the time of the original read, and the lock is maintained until the update has been completed. This ensures that no other user can obtain a lock on the item until such time as the original user has updated or released the record. The downside of this type of locking is that the lock may be maintained for a considerable period of time – at least the duration of any amendments to the record, plus distraction time. Users have been known to go out for lunch leaving a record locked on their screen – much to the annoyance of other users.

Optimistic locking works on the premise that in most cases, there will be no contention between users for a particular record. Therefore, the record is only locked immediately prior to update. When this occurs, the record on disk is compared with the original record. If the records are the same, then no one else has updated the record, and it is safe to update the record. If the record has changed, then program needs to offer choices about how to handle the situation.

The advantage of optimistic locking is that records are only locked for brief periods of time. The disadvantage is that more programming is required to handle the situation where records have changed between the original read and the read done immediately prior to update.

Pessimistic locking looks like:

```
READ record setting update lock
process record
WRITE record
```

Optimistic locking looks like

```
READ record
process record
READ record setting update lock
If the record is unchanged from the original READ then
    WRITE the updated record
Else
    do something else
END
```

An update lock is obtained by using the READU statement. This is a variant of the READ statement:

```
READU var FROM filevar, record-id {LOCKED statements} {THEN statements} {ELSE statements}
```

For example:

```
ok = @TRUE
READU cust.rec FROM customers, cust.id LOCKED
  ok = @FALSE
  MSG = 'Record ':cust.id:' is locked by user ':STATUS()
END ELSE
  cust.rec = ''
END
IF ok THEN
  GOSUB updaterec
  WRITE cust.rec ON customers, cust.id
END
```

Or:

```

READ cust.rec FROM customers, cust.id ELSE cust.rec = ''
cust.rec.orig = cust.rec
GOSUB updaterec
ok = @FALSE
tries = 0
LOOP
  tries += 1
  READU cust.rec.curr FROM customers, cust.id LOCKED
  NAP 10
END THEN
  ok = @TRUE
END ELSE
  ok = @TRUE
  cust.rec.curr = ''
END
UNTIL ok OR (tries GE 5) DO REPEAT
IF ok THEN
  IF cust.rec.curr EQ cust.rec.orig THEN
    WRITE cust.rec ON customers, cust.id
  END ELSE
    RELEASE cust.rec, cust.id
    MSG = 'Record ':cust.id:' has been changed by another user'
  END
END ELSE
  MSG = 'Could not obtain lock on record: ':cust.id
END

```

The first code fragment is an example of pessimistic locking. If the code cannot obtain the lock, then no update takes place and an error message is returned.

The second fragment is an example of optimistic locking where the record is read from the file and updated (in memory) before checking whether it is OK to write the record. Given that locks will only be held momentarily in an optimistic locking scenario, the READU is placed in a short loop. If the record is locked, then the process sleeps for 10 milliseconds before trying again. The process will loop in this manner until it successfully reads the record, or it has five unsuccessful reads. Once a lock has been obtained, the record that is read is compared with the one read prior to the update process. If the record has not been changed, then the updated record is written to the file; otherwise the lock is released and an error message returned to the user.

Note that any time that a lock is obtained, then it must be released. Locks can be released by:


- the RELEASE statement
- the WRITE statement (unless WRITEU is used which maintains the lock)
- terminating the program.

Failure to release locks could result in the system running out of locks in the lock table. To check the number of locks available, type CONFIG from the command prompt, and check the value of NUMLOCKS. The number of locks available to the system can be changed by using the Configuration Editor. See the online help for more information.

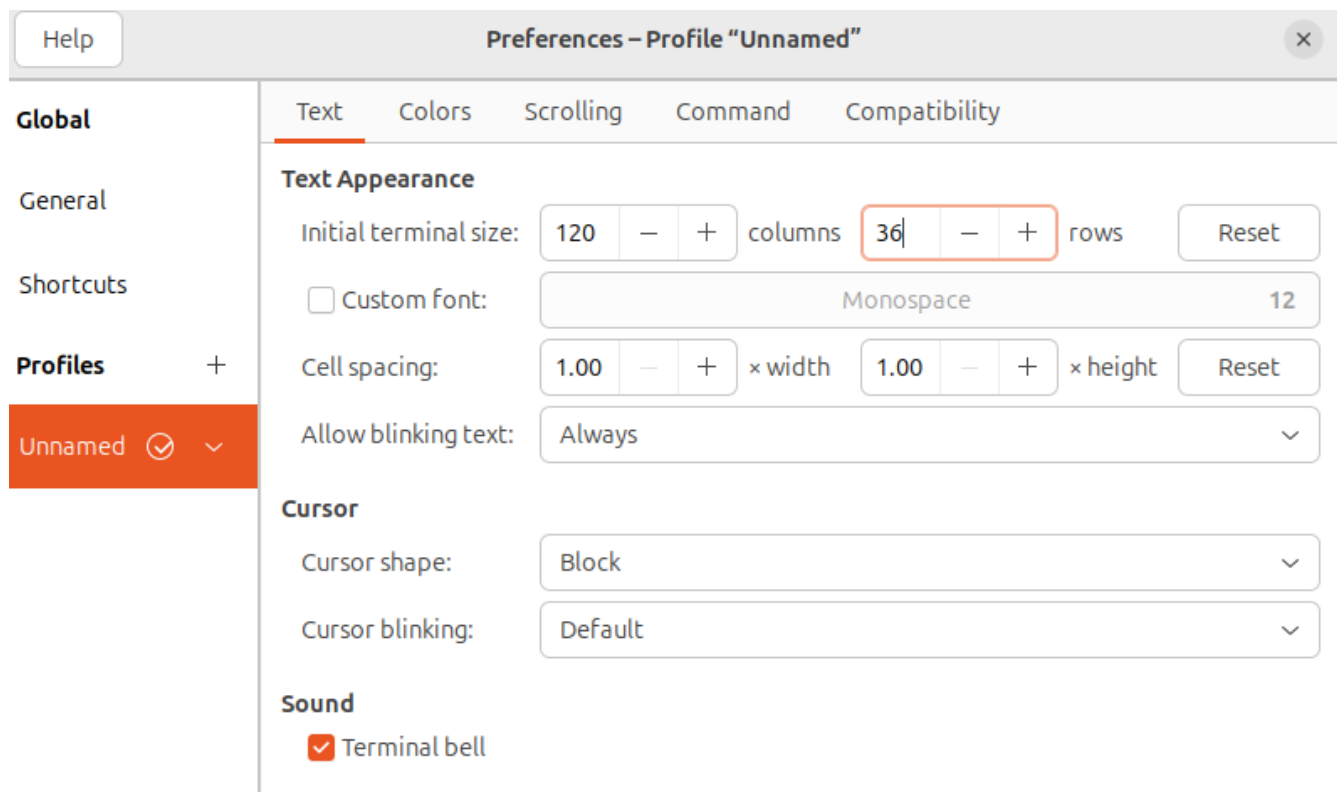
SECTION TWO Getting Started With SD

Installation - Debian 12 with Gnome or Ubuntu 24.04

This chapter will document how to install SD on Ubuntu 24.04. Open a terminal window and type “`sudo apt update && sudo apt full-upgrade`” to make sure that the OS software is current.

 When the upgrade is finished, reboot and log in again. Open a terminal window and click the hamburger menu icon (shown at left) on the right side of the title bar. Select “Preferences” from the drop down menu. In the Preferences windows, click “Unnamed” on the left side panel.

As shown in the figure below, the “Initial terminal size:” should be 120 columns by 36 rows. You may also want to select the “Custom font:” check box and change the font size.



When you have made your changes, close the window by clicking the “X” in the upper right corner of the window. Now reopen the terminal and you should see that the terminal window now has the dimensions that you asked for.

In the open terminal type the following commands:

sudo apt install git
git clone https://github.com/stringdatabase/sdb64

The source code repository will be copied from the github server and will create the sdb64 directory underneath your home directory.

Now type the following commands:

cd sdb64
./debian-installsd.sh

Press “y” and then “Enter” when asked “Continue?”

Enter your password when requested.

At the “<cr> to INSTALL “Q” to exit prompt just press “Enter”

At the “Restart computer now?” enter “y”.

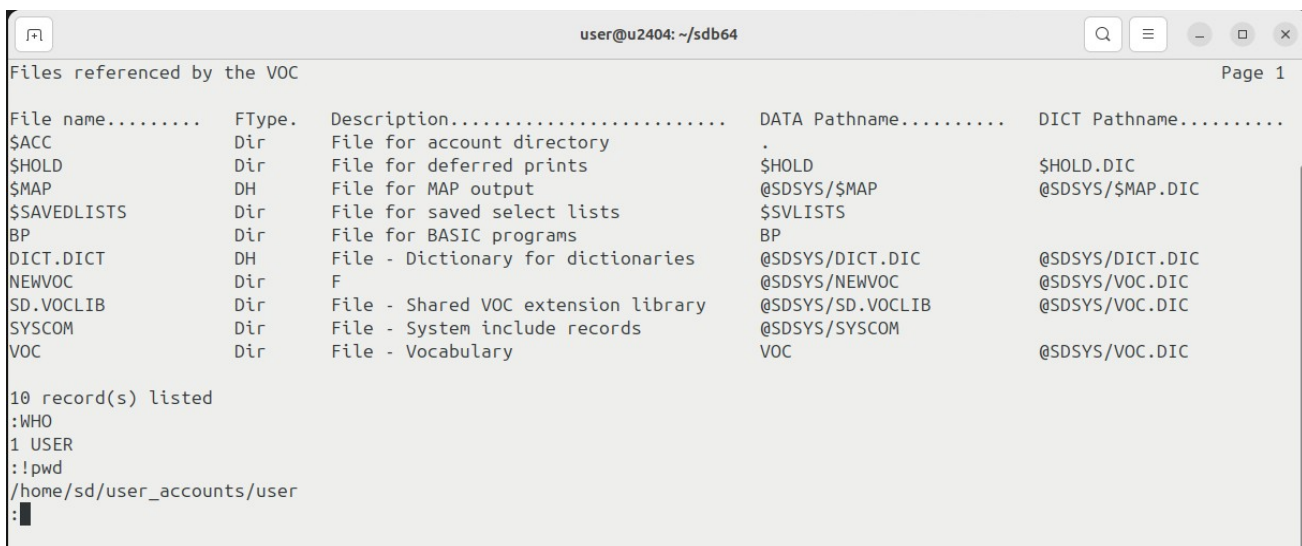
When the computer has restarted, open a terminal. Enter “sd” . At the colon prompt:

Type “**LISTF**” to see a list of the files assigned to your user account.

Type “**WHO**” to see your user name.

Type “**!pwd**” to see where your personal account directory is located.

Type “**OFF**” to exit the database and return to the operating system prompt.



```
user@u2404: ~/sdb64
Files referenced by the VOC
Page 1
File name..... FType.  Description.....  DATA Pathname.....  DICT Pathname.....
$ACC             Dir     File for account directory      .
$HOLD            Dir     File for deferred prints        $HOLD
$MAP             DH      File for MAP output             @SDSYS/$MAP
$SAVEDLISTS     Dir     File for saved select lists     $$VLISTS
BP              Dir     File for BASIC programs        BP
DICT.DICT       DH      File - Dictionary for dictionaries @SDSYS/DICT.DIC
NEWVOC          Dir     F                                @SDSYS/NEWVOC
SD.VOCLIB       Dir     File - Shared VOC extension library @SDSYS/SD.VOCLIB
SYSCOM          Dir     File - System include records   @SDSYS/SYSCOM
VOC             Dir     File - Vocabulary              VOC
DICT Pathname..... @SDSYS/DICT.DIC
DICT Pathname..... @SDSYS/VOC.DIC
DICT Pathname..... @SDSYS/VOC.DIC
DICT Pathname..... @SDSYS/VOC.DIC

10 record(s) listed
:WHO
1 USER
:!pwd
/home/sd/user_accounts/user
:
```

To log into the database as the administrator type “sudo sd” and enter your password when requested at the operating system prompt. You must have ‘sudo’ privileges on your computer to access the administrator account.

Type “**LISTF**” to see a list of files assigned to the administrator account.

Type “**WHO**” and you will see that the name of the administrator account is “SDSYS”.

Type “**!pwd**” to see where the SDSYS account directory is located.

```
user@u2404: ~/sdb64
Files referenced by the VOC Page 1
File name..... FType. Description..... DATA Pathname..... DICT Pathname.....
$ACC           Dir      File for account directory      .
$HOLD          Dir      F                                $HOLD
$MAP           Dir      File for MAP output              @SDSYS/$MAP
ACCOUNTS       Dir      F                                ACCOUNTS
BP             Dir      F                                BP
BP.OUT         Dir      F                                BP.OUT
DICT.DICT      DH       File - Dictionary for dictionaries @SDSYS/DICT.DIC
GPL.BP         Dir      F                                GPL.BP
GPL.BP.OUT     Dir      F                                GPL.BP.OUT
MESSAGES       Dir      F                                @SDSYS/MESSAGES
NEWVOC         Dir      F                                @SDSYS/NEWVOC
QFILE          DH       F                                @SDSYS/$IPC
SD.VOCLIB      Dir      File - Shared VOC extension library @SDSYS/SD.VOCLIB
SYSCOM         Dir      File - System include records     @SDSYS/SYSCOM
VOC            Dir      File - Vocabulary                 VOC
15 record(s) listed
:WHO
2 SDSYS
:!pwd
/usr/local/sdsys
:█
```

As with the personal account, enter **“OFF”** to exit to the operating system prompt.

Installation on server without a GUI

SD can also be installed on the servers that do not have a graphical user interface (GUI). Debian 12, Ubuntu 24.04 and Fedora 40 servers are supported.

The process is very similar to those discussed for the desktop versions that were covered in the preceding Chapters. Once you have logged in you can issue the following commands to install SD.

Ubuntu 24.04 Server or Debian 12

```
sudo apt update && sudo apt full-upgrade
sudo apt install git
git https://github.com/stringdatabase/sdb64
cd sdb64
./debian-installsd.sh
sudo reboot
login
“sd” = user account “sudo sd” = admin account
```

Installation on WSL

SD successfully installs on Windows Subsystem for Linux, however there are a few things to watch for:

From Google AI Overview:

If WSL is not honoring group privileges, the issue likely relates to how WSL interacts with Windows file systems (DrvFS) or requires a user session reset after adding a user to a group.

Solutions

1. Enable Linux Metadata on Windows Drives (for DrvFS) By default, files on Windows drives mounted in WSL (e.g.,) may not have full Linux metadata support, causing permission issues.

Create or edit the /etc/wsl.conf file:

from bash

```
sudo nano /etc/wsl.conf
```

Add the following lines:

```
[automount]
```

```
options = "metadata"
```

Exit and terminate WSL:

Close the WSL terminal and run the following command in a Windows PowerShell or Command Prompt with administrator privileges:

```
wsl --shutdown
```

Restart WSL

The changes should now allow ownership and permissions (using `chmod` and `chown`) to be honored.

Configuration

SD Configuration parameters are found in file /etc/sd.conf These values override the default values defined in gplsrc/config.c.

APILOGIN Ignor (0) or Require (1) User name and password on remote api connection. If set to 0, we pull username, user id and group id from peer in (start_connection). If these are populated, we ignore the passed user name (either came in as a local user using the API or as a remote user using ssh and the API). Either way we have already gone through username and password authentication. If 1 require valid user name and password to be passed in api connection method.

CREATUSR=n Allow create.account to create os user (dflt=1 yes, 0=no)

DEADLOCK=n Trap deadlocks?

DEBUG=n Debug features enabled? (bit flags)

FDS=n Set FDS limit (default is no limit)

FLTDIFF=f Wide zero value

GDI=n Select default API calls for printing

GRPDIR Group Accounts default parent directory

GRPSIZE=n Default group size when creating a dynamic file

MAXIDLEN=63 Maximum record id len

MUSTLOCK=1 Must hold update or file lock to write or delete record

NETFILES=0 Allow remote files?
 0x0001 Allow outgoing NFS
 0x0002 Allow incoming Q_M_Net

NUMFILES=n Maximum number of files open (all users)

NUMLOCKS=n Maximum number of record locks

OBJECTS=n Limit on loaded object code count (0 = no limit)

OBJMEM=n Limit on locade object size (kb, 0 = no limit)

SDCLIENT=n SDClient rules (0=all, 1=no call/exec, 2=restricted call)

SDSYS=path SDSYS directory path

SAFEDIR=1 Use careful update to directory files

SORTMEM=n Threshold for disk based sort (units of 1kb)

SORTWORK=path Pathname of sort workfile directory

STARTUP=cmd Run command on starting SD

TEMPDIR=path Pathname of temporary directory

TERMINFO=path Pathname of terminfo directory

TXCHAR=1 Enable ansi/oem character translation (default = 1)

USRDIR User Accounts default parent directory
YEARBASE=n Two digit year base (default = 1930)

Note Current system configuration values can be displayed with the CONFIG command.

```
:CONFIG
Virtual Machine Version Number 2.6-6
APILOGIN 1
CMDSTACK 99
CREATUSR 1
DEADLOCK 0
DUMPDIR
ERRLOG 50 kb
EXCLREM 0
FILERULE 0
FLTDIFF 0.00000000002910
FSYNC 0
GDI 0
GRPDIR /home/sd/group_accounts
GRPSIZE 2
INTPREC 13
JNLMODE 0
JNLDIR
LPTRHIGH 66
LPTRWIDE 80
MAXCALL 10000
MAXIDLEN 63
MUSTLOCK 0
NETFILES 0
NUMFILES 80
NUMLOCKS 100
NUMUSERS 20
OBJECTS 0 [No limit]
OBJMEM 0 [No limit]
PDUMP 0
RECCACHE 0
RINGWAIT 1
SAFEDIR 0
SDCLIENT 0
SH
SH1
SORTMEM 4096 kb
SORTMRG 4
SORTWORK /tmp
SPOOLER
STARTUP
TEMPDIR /tmp
TERMINFO
USRDIR /home/sd/user_accounts
YEARBASE 1930
```

SD Connection Methods

This chapter will document how to connect with the SD server. Connections can be “terminal” based, or via the SD API server. Note all connections to the SD server are logged to syslog, with the identifier “sd_Log”. If journalctl is installed on your linux box, the following command will display syslog messages for sd:

```
journalctl -t sd_Log
```

or

```
journalctl -t sd_Log -S today
```

Terminal Connections.

In an open terminal program type “sd”. If the current os user has been setup for SD access you will see the following:

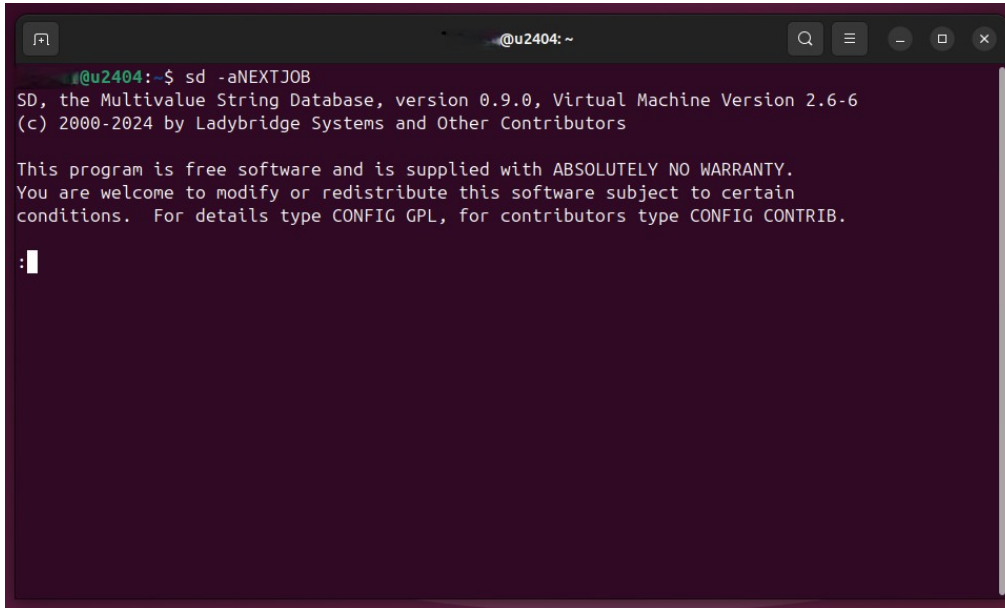
A terminal window with a dark background and light text. The window title is "@u2404: ~". The prompt is "@u2404:~\$". The user has entered the command "sd". The output is: "SD, the Multivalue String Database, version 0.9.0, Virtual Machine Version 2.6-6 (c) 2000-2024 by Ladybridge Systems and Other Contributors". Below this is a paragraph of text: "This program is free software and is supplied with ABSOLUTELY NO WARRANTY. You are welcome to modify or redistribute this software subject to certain conditions. For details type CONFIG GPL, for contributors type CONFIG CONTRIB." The prompt is now ":".

```
@u2404:~$ sd
SD, the Multivalue String Database, version 0.9.0, Virtual Machine Version 2.6-6
(c) 2000-2024 by Ladybridge Systems and Other Contributors

This program is free software and is supplied with ABSOLUTELY NO WARRANTY.
You are welcome to modify or redistribute this software subject to certain
conditions. For details type CONFIG GPL, for contributors type CONFIG CONTRIB.

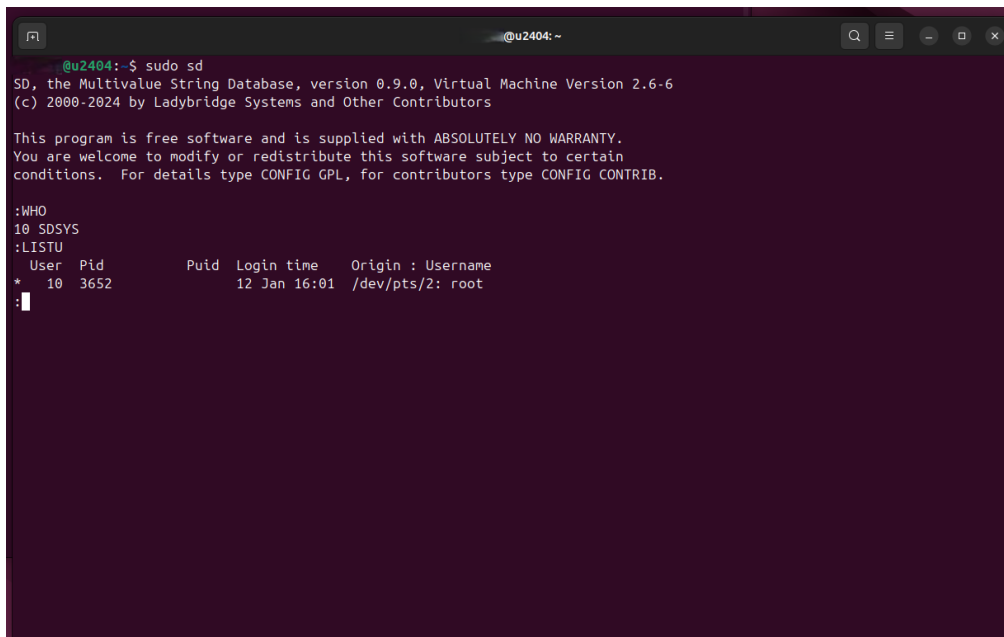
:
```

To access a group account (that the user has access to, see CREATE.ACCOUNT) use the -a option:



```
@u2404: ~  
@u2404:~$ sd -aNEXTJOB  
SD, the Multivalue String Database, version 0.9.0, Virtual Machine Version 2.6-6  
(c) 2000-2024 by Ladybridge Systems and Other Contributors  
  
This program is free software and is supplied with ABSOLUTELY NO WARRANTY.  
You are welcome to modify or redistribute this software subject to certain  
conditions. For details type CONFIG GPL, for contributors type CONFIG CONTRIB.  
:  
|
```

Access to the SDSYS account requires root privileges. Login via sudo sd.



```
@u2404: ~  
@u2404:~$ sudo sd  
SD, the Multivalue String Database, version 0.9.0, Virtual Machine Version 2.6-6  
(c) 2000-2024 by Ladybridge Systems and Other Contributors  
  
This program is free software and is supplied with ABSOLUTELY NO WARRANTY.  
You are welcome to modify or redistribute this software subject to certain  
conditions. For details type CONFIG GPL, for contributors type CONFIG CONTRIB.  
  
:WHO  
10 SDSYS  
:LISTU  
User Pid Puid Login time Origin : Username  
* 10 3652 12 Jan 16:01 /dev/pts/2: root  
:  
|
```

API Connections.

There are two methods to connect to the API server using the sdclilib library.

SDConnectLocal(account)

The local connection runs as the user of the process that invoked the SDConnectLocal method. The sdclilib library creates a fork of the existing process and uses the c function execl() to replace the newly created child process with the sd program running with the -C and -Q command options. The -C option tells sd to established pipes for communication between the parent process and the child process created with the fork(). The -Q option tells sd to run the API server program APISVR.

SDConnect(host, port, username, password, account)

Remote connections are made with the SDConnect method. ALL remote API connections are via an AF_UNIX socket located at “/tmp/sdsys/sdclient.socket” on the SD server.

This requires the following setup:

An sshd server needs to be installed on the SD server.

An ssh client needs to be installed on the client wishing to connect to the SD server.

Example ssh command:

Assume the SD server is running on a box with the network name Z400.

Issue the following command in an open terminal window on the client side:

```
ssh -L 4245:/tmp/sdsys/sdclient.socket -N <username>@Z400
```

- <username> needs to be defined on the SD server and setup as an sd user.

On the first time establishing the ssh tunnel, you should see something like:

```
$ ssh -L 4245:/tmp/sdsys/sdclient.socket -N <username>@Z400 <== entered ssh command
```

```
The authenticity of host 'z400 (127.0.1.1)' can't be established.  
ED25519 key fingerprint is  
SHA256:+uvamVBTjIN0OF4loZUAPgtgYxV5bCbWASBbZZTZR4g.  
This key is not known by any other names.  
Are you sure you want to continue connecting (yes/no/[fingerprint])? <== answer “yes”  
Warning: Permanently added 'z400' (ED25519) to the list of known hosts.
```

```
<username>@z400's password: <== enter the password for <username>
```

Once you enter the password the terminal will just "hang" (its running the ssh client).

You should be able to verify the tunnel on the SD server by issuing the following:

```
% sudo lsof -i -n | egrep '\<ssh\>
```

In the SDConnect method you would specify port 4245. Note: the default port in the sdclilib.so SDConnect method is 4245, if you use -1 for port number in the SDConnect call it will use port 4245.

The following is a connection example using Python and the sdclilibwrap.py wrapper found in the install download in folder ../sdb64/sd64/examples/python/python_api_test

Note: This wrapper currently does not wrap all functions in sdclilib.so.

For simplicity, copy sdclilibwrap.py and the compiled library sdclilib.so to a test folder.

Start a terminal window and execute the ssh command.

Start a terminal window on the server and navigate to the test folder, start up python.

Note in the example use valid values for: <username>, <password>, <account>

```
xxxx@u2404:~/python_stuff$ python3
Python 3.12.3 (main, Nov 6 2024, 18:32:19) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sdclilibwrap as sdmelib
>>> sdmelib.sdmeInitialize()
>>> status = sdmelib.sdmeConnect("127.0.0.1", -1, "<username>", "<password>", "<account>")
>>> print(status)
1
>>> results, err = sdmelib.sdmeExecute("who")
>>> print(results)
14 <username>
>>> results, err = sdmelib.sdmeExecute("listfl")
>>> print(results)
```

Files local to this account referenced by the VOC

Page 1

File name.....	FType	Description.....	DATA Pathname.....	DICT Pathname.....
\$ACC	Dir	File for account directory	.	
\$HOLD	Dir	File for deferred prints	\$HOLD	\$HOLD.DIC
\$MAP	DH	File for MAP output	@SDSYS/\$MAP	@SDSYS/\$MAP.DIC
\$SAVEDLISTS	Dir	File for saved select lists	\$SVLISTS	
BP	Dir	File for BASIC programs	BP	
BP.OUT	Dir	F	BP.OUT	
CUSTOMERS	DH	F	CUSTOMERS	CUSTOMERS.DIC
DICT.DICT	DH	File - Dictionary for dictionaries	@SDSYS/DICT.DIC	@SDSYS/DICT.DIC
DTST	Dir	F	DTST	DTST.DIC
MYTEST	DH	F	MYTEST	MYTEST.DIC
NEWVOC	Dir	F	@SDSYS/NEWVOC	@SDSYS/VOC.DIC
REPORTS	Dir	F	REPORTS	REPORTS.DIC
REPORTS_DEF	Dir	F	REPORTS_DEF	REPORTS_DEF.DIC
SD.VOCLIB	Dir	File - Shared VOC extension library	@SDSYS/SD.VOCLIB	@SDSYS/VOC.DIC
SYSCOM	Dir	File - System include records	@SDSYS/SYSCOM	
TESTDATA	DH	F	TESTDATA	TESTDATA.DIC
VOC	Dir	File - Vocabulary	VOC	@SDSYS/VOC.DIC
prefix_tst	DH	F	PREFIX_TST	PREFIX_TST.DIC

```
18 record(s) listed
>>> print (sdmelib.sdmeConnected())
1
>>> sdmelib.sdmeDisconnect()
>>> print (sdmelib.sdmeConnected())
0
>>> quit()
```

SD User and Group Accounts.

Accounts are created with the CREATE.ACCOUNT command. This command can only be issued by the system admin, logged into the sdsys account via “sudo sd”.

SD user account default parent directory: /home/sd/user_accounts

SD group account default parent directory: /home/sd/group_accounts

CREATE.ACCOUNT USER <username> {NO.QUERY}

account created in: /home/sd/user_accounts/<username>
owner : group set to <username> : ”sdu_”<username>,
permissions set to 775

If the OS user <username> does not exist, it is created by this command.

Please note the convention is for all Account Names to be upper case and all OS User Names to be lower case.

CREATE.ACCOUNT will automatically perform the case conversions.

CREATE.ACCOUNT GROUP <grp acct name> {NO.QUERY}

account created in /home/sd/group_accounts/<grp acct name>
owner : group set to sdsys : ”sdg_”<grp_acct_name>
permission set to 775

Please note the convention is for all Account Names to be upper case and all OS Group Names to be lower case.

CREATE.ACCOUNT will automatically perform the case conversions.

Note: users will need to be added to accounts with:

MODIFY.ACCOUNT <account name> <ADD/DELETE> <username>

The CREATE.ACCOUNT command creates an entry in /etc/group for each account it creates. For USER accounts the group name will be “sdu_<username>” for GROUP accounts the group name will be ”sdg_” <grp_acct_name>.

* Note: sd user and group accounts have the setgid bit set on the parent directory. When the bit is set for a directory, **the set of files in that directory will have the same group as the group of the parent directory, and not that of the user who created those files.** This is used for file sharing since the files can now be modified by all the users who are part of the group of the parent directory.

CREATE.ACCOUNT OTHER acc.name pathname {NO.QUERY} - account created in pathname, maintained for backwards compatibility. This will require manual editing of the accounts record

to specify linux group the account belongs to.
User should also set the setgid bit of account's parent directory.

Notes:

CREATE.ACCOUNT saves the group name in fld 3 of ACCOUNTS record (for both CREATE.ACCOUNT USER and GROUP).

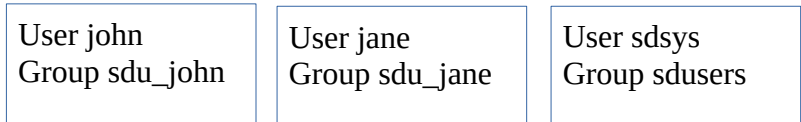
LOGIN and LOGTO(CPROC) checks to confirm the user is a member of sdsusers group and the ACCOUNT group before allowing access to the account.

CREATE.ACCOUNT adds users root, sdsys, (and <username> for account types USER) to the entry created in /etc/group. **However user membership to the group does not immediately come into effect for the process creating the account.** A Linux logout and login is required. Signs of this issue are errors when performing file actions on the new account.

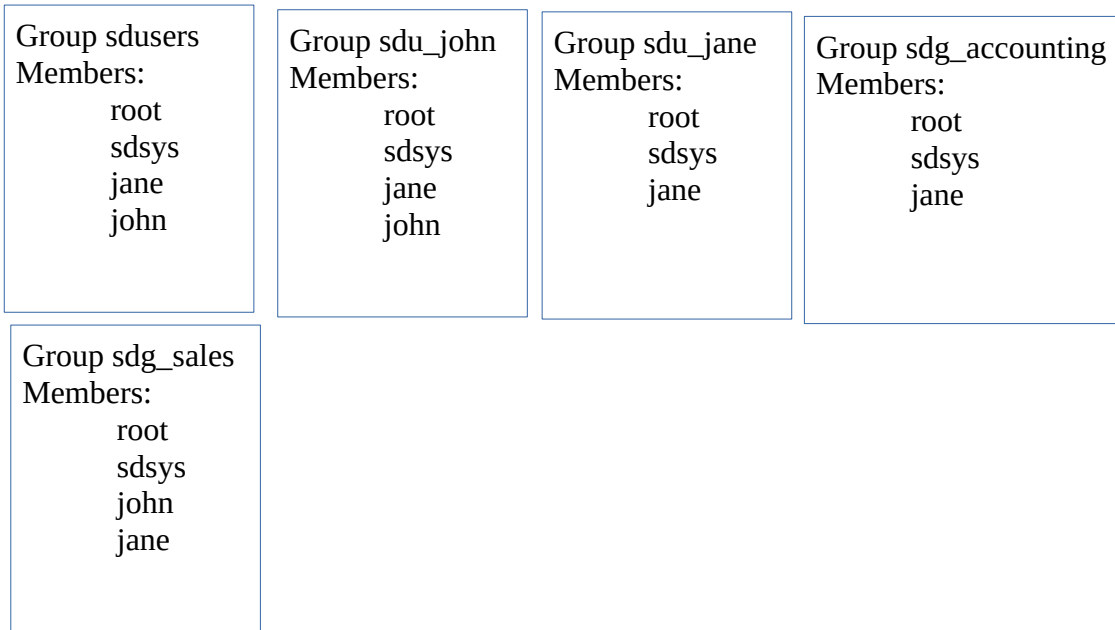
Example: Error 8207 creating file at line 416 of \$CREATEF when using CREATE.FILE.

Example: On a system with 4 sd accounts setup (john, jane, accounting and sales)

Linux Users



Linux Groups



With the above linux group membership the following users are allowed access to the SD Account specified

SD Account JANE Group: sdu_jane Allowed Users: root sdsys jane	SD-Account JOHN Group: sdu_john Allowed Users: root sdsys john jane	SD-Account ACCOUNTING Group: sdg_accounting Allowed Users: root sdsys jane	SD-Account SALES Group: sdg_sales Allowed Users: root sdsys jane john
---	---	---	---

Parent Directories:

linux file permission setting

Directory	Account Name	Owner	Group	other
/home/sd/user_accounts		sdsys (rwx)	sdusers (rwx)	(r_x)
/home/sd/user_accounts/jane	JANE	jane (rwx)	sdu_Jane (rwx)	(r_x)
/home/sd/user_accounts/john	JOHN	john (rwx)	sdu_John (rwx)	(r_x)
/home/sd/group_accounts/accounting	ACCOUNTING	sdsys (rwx)	sdg_accounting (rwx)	(r_x)
/home/sd/group_accounts/sales	SALES	sdsys (rwx)	sdg_sales (rwx)	(r_x)

Notes:

User root has access to all accounts.

User sdsys has access to all accounts via sd. On entry to sd via sudo sd, the effective user is changed to sdsys, with the ability to return to user root as necessary (controlled in CPROC).

User jane has access to accounts JANE, JOHN, ACCOUNTING and SALES.

User john has access to accounts JOHN, SALES

TCL – The Command Line

TCL Commands Available in SD - Usage the same as in OpenQM 2.6.6

- Removed Commands and New Commands Unique to SD are listed at bottom of this page

*	Comment
\$ECHO	Paragraph tracing
!	Synonym for SH
ABORT	Abort processing and return to command prompt
ALIAS	Create a temporary alias for a command
ANALYSE.FILE	Analyse structure and usage of dynamic file
ANALYZE.FILE	Synonym for ANALYSE.FILE
AUTOLOGOUT	Set inactivity timer
BASIC	Compile SDBasic programs
BELL	Enable or disable audible alarm
BUILD.INDEX	Build an alternate key index
CATALOG	Synonym for CATALOGUE
CATALOGUE	Add program to system catalogue
CD	Synonym for COMPILE.DICT
CLEARDATA	Synonym for CLEAR.DATA
CLEARINPUT	Synonym for CLEAR.INPUT
CLEARPROMPTS	Synonym for CLEAR.PROMPTS
CLEARSELECT	Synonym for CLEAR.SELECT
CLEAN.ACCOUNT	Remove records from \$HOLD, \$COMO and \$SAVEDLISTS
CLEAR.ABORT	Clear the abort status in an ON.ABORT paragraph

CLEAR.DATA	Clear the data queue
CLEAR.FILE	Remove all records from a file
CLEAR.INPUT	Clear keyboard type-ahead
CLEAR.LOCKS	Release task locks
CLEAR.PROMPTS	Clear inline prompt responses
CLEAR.SELECT	Clear one or all select lists
CLEAR.STACK	Clear the command stack
CLR	Clear display
CNAME	Rename a file or record within a file
COMO	Activate or deactivate command output files
COMPILE.DICT	Compile I-types in a dictionary
CONFIG	Display configuration parameters
CONFIGURE.FILE	Change file configuration parameters
COPY	Copy records
COPY.LIST	Copy a saved select list
COUNT	Count records
CREATE.FILE	Create a file
CREATE.INDEX	Create an alternate key index
CS	Synonym for CLR
CT	Display records from a file
DATA program	Add text to the data queue for associated verb or program
DATE	Display the date and time
DATE.FORMAT	Selects default date format
DEBUG	Debug SDBasic program
DELETE	Delete records from a file
DELETE.CATALOG	Synonym for DELETE.CATALOGUE
DELETE.CATALOGUE	Delete a program from the system catalogue
DELETE.COMMON	Delete a named common block

DELETE.FILE	Delete a file
DELETE.INDEX	Delete an alternate key index
DELETE.LIST	Delete a saved select list
DISPLAY	Display text
DUMP character format	Display records from a file in hexadecimal and character format
ECHO	Disable or enable keyboard echo
ED	Line editor
EDIT	Synonym for ED
EDIT.LIST	Edit a saved select list
FORMAT	Apply conventional formatting to a SDBasic program
FORM.LIST	Create a select list from a file record
FSTAT	Collect and report file statistics
GENERATE	Generate a SDBasic include record from a dictionary
GET.LIST	Retrieve a previously saved select list
GET.STACK within a paragraph	Restore a saved command stack GO Jump to a label within a paragraph
HSM	Hot Spot Monitor performance monitoring tool
HUSH execution in paragraphs	Disable or enable display output IF Conditional execution in paragraphs
LIST	List records from a file
LIST.COMMON	List named common blocks
LIST.DIFF	Form difference of two saved select lists
LIST.FILES	List details of open files
LIST.INDEX	List details of an alternate key index
LIST.INTER	Form intersection of two saved select lists
LIST.ITEM	List records from a file in internal format
LIST.LABEL	List records from a file in address label format
LIST.LOCKS	List task lock status
LIST.READU	List file, read and update locks

LIST.UNION	Form union of two saved select lists
LIST.VARS	List user @-variables
LISTF	List all files defined in the VOC
LISTFL	List all local files defined in the VOC
LISTFR	List all remote files defined in the VOC
LISTK	List all keywords defined in the VOC
LISTPA	List all paragraphs defined in the VOC
LISTPH	List all phrases defined in the VOC
LISTQ	List all indirect file references in the VOC
LISTR	List all remote items defined in the VOC
LISTS	List all sentences defined in the VOC
LISTU	List users currently in SD
LISTV	List all verbs defined in the VOC
LOCK	Set a task lock
LOGMSG	Write a message to the error log
LOGOUT	Terminate a phantom process
LOGTO	Change to an alternative account
LOOP / REPEAT	Defines loop within paragraph
MAKE.INDEX	Create and build an alternate key index
MAP	Display a list of the catalogue contents
MERGE.LIST	Create a select list by merging two other lists
MESSAGE	Send a message to selected other users
MODIFY	Modify records in a file
NLS	Set or report national language support values
NSELECT	Remove items from a select list
OFF	Synonym for QUIT
OPTION	Set, clear or display options
PAUSE	Display "Press return to continue" prompt
PDEBUG	Runs the phantom debugger

PDUMP	Generate a process dump file
PHANTOM	Initiate a background process
PRINTER	Administer print units
PSTAT	Report process status
PTERM	Set or display terminal characteristics
QSELECT selected records	Construct a select list from the content of selected records
QUIT	Terminate session or revert to lower command level
RELEASE	Release record or file locks
RENAME	Synonym for CNAME
REPORT.SRC	Display @SYSTEM.RETURN.CODE at command prompt
REPORT.STYLE	Sets the default style for query processor reports
RUN	Run a compiled SDBasic program
SAVE.LIST stack	Save a select list SAVE.STACK Save the command stack
SEARCH	Search file for records containing string(s)
SELECT	Select records meeting criteria
SED	Full screen editor
SET	Set a user @variable
SET.DATE	Set SD processing date
SET.EXIT.STATS	Set final exit status value
SET.FILE	Set a Q-pointer to a remote file
SET.TRIGGER dynamic file	Set, remove or display trigger function for a dynamic file
SETPTR	Set print unit characteristics
SH	Execute shell command
SHOW	Build select list interactively
SLEEP	Suspend process until specified time
SORT	List records sorted by record key

SORT.ITEM format	List records sorted by record key in internal format
SORT.LABEL record key	List records in address label format, sorted by record key
SP.CLOSE	Close a print unit previously in "keep open" mode
SP.OPEN	Open a print unit in "keep open" mode
SP.VIEW	View and print records from \$HOLD or other files
SPOOL	Send record(s) to the printer
SSELECT record key	Select records meeting criteria, sorting list by record key
STATUS	Display list of active phantom processes
STOP	Terminate an active paragraph
SUM	Report total of named fields
TERM	Set or display terminal window type and size
TIME	Display date and time
UNLOCK	Unlock a record or file
UPDATE.ACCOUNT	Update VOC items from NEWVOC
UPDATE.RECORD	Utility to update records in file
WHO	Display user number and account name
WHERE	Display pathname of current account

 Commands Unique to SD - Only available to administrators

CREATE.ACCOUNT	Make a new SD account
DELETE.ACCOUNT	Delete a SD account
MODIFY.ACCOUNT	Modify an existing SD account

Commands Unique to SD - Available to all users

MICRO Use OS micro editor to edit records

Encryption Commands Not available in SD
- source code for CRYPTO was not provided in original GPL release

CREATE.KEY
DELETE.KEY
ENCRYPT.FILE
GRANT.KEY
LIST.KEYS
RESET.MASTER.KEY
REVOKE.KEY

TCL Commands available in SD if the optional TAPE and RESTORE
subsystem is installed

ACCOUNT.RESTORE
FIND.ACCOUNT
SEL.RESTORE
SET.DEVICE Attach a tape device
T.ATT
T.DET

T.DUMP
T.EOD
T.FWD
T.LOAD
T.RDLBL
T.READ
T.REW
T.STAT
T.WEOF

Other OpenQM 2.6.6 TCL Commands not available in SD

ADMIN.USER
BLOCK.PRINT
BLOCK.TERM
CREATE.USER
DELETE.USER
HELP
LIST.USERS
LISTM
LISTPQ
LOGIN.PORT
MED
PASSWORD
PTERM
RESTORE.ACCOUNTS
SCRB

SECURITY

SET. ENCRYPTION. KEY. NAME

SETPORT

SET. QUEUE

SP. ASSIGN

UPDATE. LICENCE

Encryption in SD

Encryption, Decryption and Encoding are handled by the libsodium package <https://doc.libsodium.org>

Calling syntax:

encrypted_text = **SDENCRYPT(Data, KeyToUse, Encoding)**

where:

- Data** = string to encrypt
- KeyToUse** = Encoded string which will decode to a crypto_secretbox_KEYBYTES bytes sized key (currently defined as 32 bytes / 256 bits).
- Encoding** = one of:
 - SD_EncodeHX** - The key and returned encrypted text are Hex Encoded (passed key is a 64 Character Hex encoded string which will be converted to 32 bytes, encrypted text will be a Hex encoded character String 2X the length of the passed string to encrypt).
 - SD_Encode64** - The key and returned encrypted text are Base64 Encoded (passed key is a 44 Character Base64 encoded string which will be converted to 32 bytes, encrypted text will be a Base64 encoded character String).

decrypted_text = **SDDECRYPT(Data, KeyToUse, Encoding)**

where:

- Data** = encrypted string
- KeyToUse** = Encoded string which will decode to a crypto_secretbox_KEYBYTES bytes sized key (currently defined as 32 bytes / 256 bits).
- Encoding** = one of:
 - SD_EncodeHX** - The key and passed encrypted text are Hex Encoded (passed key is a 64 Character Hex string which will be converted to 32 bytes, encrypted text will be a Hex Character String 2X the length of the returned decrypted text).
 - SD_Encode64** - The key and passed encrypted text are Base64 Encoded

Why libsodium ?

From <https://doc.libsodium.org/>

Sodium is a modern, easy-to-use software library for encryption, decryption, signatures, password hashing, and more.

Encryption, Decryption and encoding using libsodium package:

<https://doc.libsodium.org/>

This routine (and doc) is based on information found here:

https://doc.libsodium.org/secret-key_cryptography/secretbox

https://github.com/jedisct1/libsodium/blob/master/test/default/secretbox_easy2.c

https://doc.libsodium.org/password_hashing/default_phf

SD encrypts using `crypto_secretbox_easy`:

```
int crypto_secretbox_easy(unsigned char *c, const unsigned char *m,  
                          unsigned long long mlen, const unsigned char *n,  
                          const unsigned char *k)
```

The `crypto_secretbox_easy()` function encrypts a message **m** whose length is **mlen** bytes, with a key **k** and a nonce **n**.

k should be `crypto_secretbox_KEYBYTES` bytes (currently defined as 32 bytes / 256 bits) and **n** should be `crypto_secretbox_NONCEBYTES` bytes.

c should be at least `crypto_secretbox_MACBYTES + mlen` bytes long.

This function writes the authentication tag, whose length is `crypto_secretbox_MACBYTES` bytes, in **c**, immediately followed by the encrypted message, whose length is the same as the plaintext: **mlen**.

Functions returning an `int` return 0 on success and -1 to indicate an error.

REM:

A 128-bit (16 byte) key can be expressed as a hexadecimal string with 32 characters.
It will require 24 characters in base64.

A 256-bit (32 byte) key can be expressed as a hexadecimal string with 64 characters.
It will require 44 characters in base64.

Note we return the encryption output with the nonce appended to the end
Rem to encode either base64 or hex before returning to SD!

SD decrypts using `crypto_secretbox_open_easy`:

```
int crypto_secretbox_open_easy(unsigned char *m, const unsigned char *c,  
                              unsigned long long clen, const unsigned char *n,  
                              const unsigned char *k);
```

c is a pointer to an authentication tag + encrypted message combination,

as produced by `crypto_secretbox_easy()`.

clen is the length of this authentication tag + encrypted message combination.

Put differently, `clen` is the number of bytes written by `crypto_secretbox_easy()`, which is `crypto_secretbox_MACBYTES` + the length of the message.

The nonce **n** and the key **k** have to match those used to encrypt and authenticate the message.

The function returns -1 if the verification fails, and 0 on success.

On success, the decrypted message is stored into `m`.

If the user wishes to use a password for encryption / decryption we need to generate a key for it.

The project recommends using `crypto_pwhash` to convert a password to a key, but to be reproducible the routine requires:

The salt to be known

Values for `opslimit` and `memlimit`

https://doc.libsodium.org/key_derivation and https://doc.libsodium.org/password_hashing/default_phf

To do this we will use function:

```
int crypto_pwhash(unsigned char * const out,  
                 unsigned long long outlen,  
                 const char * const passwd,  
                 unsigned long long passwdlen,  
                 const unsigned char * const salt,  
                 unsigned long long opslimit,  
                 size_t memlimit, int alg);
```

The `crypto_pwhash()` function derives an **outlen** bytes long key from a password **passwd** whose length is **passwdlen** and a **salt** whose fixed length is `crypto_pwhash_SALTBYTES` bytes.

`passwdlen` should be at least `crypto_pwhash_PASSWD_MIN` and `crypto_pwhash_PASSWD_MAX`.

`outlen` should be at least `crypto_pwhash_BYTES_MIN` = 16 (128 bits) and at most `crypto_pwhash_BYTES_MAX`.

The salt should be unpredictable. `randombytes_buf()` is the easiest way to fill the `crypto_pwhash_SALTBYTES` bytes of the salt.

Keep in mind that to produce the same key from the same **password**, the same algorithm, the same **salt**, and the same values for **opslimit** and **memlimit** must be used.

SD provides two subroutines to support password to key creation:

!SD_GET_SALT(mysalt) - creates a base64 encoded salt and returns in `mysalt`.

!SD_KEY_FROM_PW(my passphrase, mysalt, mykey) – creates a base64 encoded key returned in `mykey`. Where:

my passphrase is a text password / phrase

mysalt – is a base64 encoded salt created via `!SD_SALT()`*

*This setup requires the user to save the salt used for key generation.

There is an example of usage found in BP/SD_ENCRYPT_EXT

Embedded Python in SD

SD from version 0.9-2 on includes the following embedded python functions. These functions are found in GPL.BP and provide the BASIC interface to the python interpreter.

PY_INITIALIZE()
calls api function Py_Initialize() - start the python interpreter

PY_IS_INITIALIZED()
calls api function Py_IsInitialized() - Is python interpreter initialized?

PY_RUNSTRING(py_script)
calls api function PyRun_String() - run a python script from string

PY_RUNFILE(py_script_file)
calls api function PyRun_File() - run a python script from file

PY_GETATTR(objname)
calls api function PyMapping_GetItemString() - access the value of a python object.

PY_FINALIZE()
calls api function Py_Finalize()

PY_LENGTH(objectname)
return python object's length (size)

PY_TYPE(objectname)
return python object's type

PY_CREATEDICT(dictname)
create python dictionary object

PY_CLEAR_DICT(dictname)
clear python dictionary object via api call PyDict_Clear().

PY_DICTVALGETS(dictname,key)
get dictionary value for item key, returned as string

PY_DICTVALSETS(dictname,key,value)
set (update) or create dictionary value for key.

PY_DICTIDEL(dictname,key)
delete dictionary key / value

PY_DICTGETKEYS(dictname)
return list of dictionary keys / value

PY_DICTGETVALUES(dictname)
return list of dictionary values

PY_STRSET(strname,value)
set (update) or create string object strname with string value.

```
PY_STRGET(strname)
    get string value of string strname

PY_LISTGETS(listname)
    return list of list items

PY_LISTAPPEND(listname,objectname)
    append objectname to list listname
```

Important Note:

Once the python interpreter is initialized within SD, it stays active until a PY_FINALIZE is issued. On exit SD will execute PY_FINALIZE if the python interpreter was previously initialized. There seems to be an issue with Initializing and Finalizing the interpreter multiple times within a process. Therefore, if you are sure an SD process is done with the python interpreter and will not re issue a PY_INITIALIZE, it is safe to call PY_FINALIZE (and free up memory taken by the Python interpreter) otherwise allow SD to perform the PY_FINALIZE on exit.

In order to use these functions in a BASIC program, a DEFFUN statement for each function must be included in the BASIC source. The SDPYFUNC.H include file found in SYSCOM provides the DEFFUN statements.

Remember strings are UTF-8 in python and must be converted to bytes (decode in encode out). Good reference here: <https://nedbatchelder.com/text/unipain.html>

There are three sample programs PY_TEST, PY_TERM (both in sdsys/ BP) and PY_GUI_TEST (in examples/python/embedded_python)

PY_TEST

PY_TEST is a BASIC program which test the python functions listed above.

PY_TERM

PY_TERM very simple python interface via terminal

PY_GUI_TEST

Program creates a simple gui dialog via a script file executed from within SD. User populates fields, hits ok, program prints entered data back out.

Uses FreeSimpleGUI which must be installed prior to running the program.
<https://github.com/spyoungtech/FreeSimpleGUI>

Script file can be found in example/python/embedded_python/sdguitest.py.

edit PY_GUI_TEST line for correct location of python script file.

```
script_file = "/home/xyz/python_stuff/sdguitest.py"
```

NOTE

This program cannot be run from root unless you have played with your systems default settings. Either copy the program to your BP directory and compile, or compile and catalog global, running from your user account.

There is also a report writer, **SD Quick Report**, which is a simple spreadsheet based report design and creation package written to test SD's embedded python.

SECTION THREE Developer Notes

Changing Revision

The following source files must be modified to correctly roll the sd revision level:

gplsrc - revstamp.h

change the following defines

```
#define MAJOR_REV 1
#define MINOR_REV 0
#define BUILD 0
#define SD_REV_STAMP "1.0-0"
```

GPL.BP - REVSTAMP.H

change the following defines

```
$define MAJOR.REV 1
$define MINOR.REV 0
$define BUILD 0
$define SD.REV.STAMP "1.0-0"
```

NEWVOC - \$RELEASE

change line 2 to match the SD_REV_STAMP from the above

```
X
1.0-0
```

VOC_TEMPLATE - \$RELEASE

change line 2 to match the SD_REV_STAMP from the above

```
X
1.0-0
```