FILETAB

Developer's Guide

S-08-09-001 Issue 1

July 2001

The National Computing Centre Limited
Oxford House, Oxford Road
Manchester M1 7ED

	UK	International
Telephone:	0161 228 6333	(+44) 161 228 6333
Fax:	0161 236 9877	(+44) 161 236 9877

C.ISAM and Informix are registered trademarks of Informix Software Inc.

SPARC is a registered trademark of Oracle Corporation

ORACLE is a registered trademark of Oracle Corporation

INGRES is a registered trademark of INGRES Corporation

Hewlett-Packard is a registered trademark of Hewlett-Packard Ltd.

FILETAB is a registered trademark of The National Computing Centre Ltd.

Linux is a registered trademark of Linus Torvalds

Microsoft Windows, Windows NT Server and Windows NT Workstation are registered trademarks of Microsoft Corporation

InstallShield is a registered trademark and service mark of InstallShield Software Corporation in the US and/or other countries

© The National Computing Centre Limited 2001
All Rights Reserved.
The copyright in this document is vested in The National Computing Centre Limited. The document must not be reproduced, by any means in whole or in part or used for manufacturing purposes, except with the prior written permission of The National Computing Centre Limited and then only on condition that this notice is included in any such reproduction.
Information contained in this document is believed to be accurate at the time of publication but no liability whatsoever can be accepted by The National Computing Centre Limited arising out of any use made of this information.
The software described in this document is supplied under a licence agreement. The software may be used or copied only in accordance with the specified terms of the licence agreement.

Issue Record

Issue	Date	Comment
1	30.07.01	Issued

Contents

Chapter 1 Introduction 1-1

About FILETAB 1-1

Programming 1-1

Report Generation 1-1

About your FILETAB Documentation 1-2

Compiling and Running FILETAB Programs 1-4

On UNIX and Linux 1-4

To Compile a FILETAB for Windows NT Program 1-4

To Run a Compiled FILETAB for Windows NT Program 1-4

Principles. 1-5

Source Statements 1-5

Fixed Logic 1-5

Example Files 1-6

Conventions Used in this Guide 1-7

Chapter 2 Simple Reports 2-1

Preliminary Notes 2-1

Sequence of Directives 2-4

Operands 2-4

Detail Statements (Detail Lines) 2-4

Comments 2-4

Length of Lines 2-5

Fixed Logic 2-6

Notes 2-10

*PROGRAM Directive 2-10

*FILE Definition. 2-10

Examples 2-11

*DICTIONARY Definition. 2-12
Record and Work Areas. 2-12
Format of *DICTIONARY Detail Statements 2-13
Examples of *DICTIONARY Detail Statements 2-14
Notes on *DICTIONARY 2-14
*INLIST Definition. 2-15
Control, Totalling and Transfer Fields. 2-15
Example 2-15
Field Specifying Characters (FSCs). 2-16
Format of *INLIST Detail Statements. 2-17
Example 2-17
Example 2-17
*INLIST Area 2-18
Input Specification Summary 2-18
Example 2-18
Notes 2-19
Printing Headings 2-19
General. 2-19
Format of *HEAD and *TITLE Directives 2-20
Including Date, Time, Page No. and Mark No. in Headings 2-20
Example 2-21
Including Data in Headings 2-21
Notes 2-21
Operands on *HEAD Directive 2-22
Printing Data 2-23
General. 2-23
*OUT Detail Statements 2-23
Example 2-24
Control and Transfer Fields 2-24
Totalling Fields - Editing Characters. 2-25
Totalling Fields - Floating Symbols 2-25
Incorporation of Literals in *OUT Detail Statements 2-25
*OUT Operands. 2-25
Examples. 2-26
Output Specification Summary 2-26

*SORT Directive. 2-27
Examples 2-28
Operation of the Sort 2-28
*END, *GO and *STOP Directives 2-29
Worked Example. 2-29
Chapter 3 Decision Tables 3-1
Decision Table Theory 3-1
Examples 3-1
Example 1 3-1
Example 2 3-4
ELSE Rule 3-4
Example 3 3-5
Example 4 3-5
Example 5 3-6
Types of Decision Table 3-6
Example 6 3-6
Initial Actions. 3-6
Example 7 3-7
Linking Decision Tables 3-7
Go To 3-7
Call 3-7
If 3-8
Recursion 3-8
Constructing Decision Tables 3-8
Decision Tables in FILETAB 3-9
Facilities and Uses of Decision Tables in FILETAB. 3-10
Interpretation of FILETAB Decision Tables 3-10
Incomplete Decision Tables. 3-12
Decision Table Format 3-13
The *DETAB Directive 3-13
Decision Table Detail Format 3-13
Decision Table Stub Syntax - Limited Entry. 3-15
Condition Operators. 3-16
Condition Verbs. 3-16
Action Operators 3-16

Action Verbs 3-17
Decision Table Entry Syntax - Limited Entry 3-19
Example 3-19
Decision Table Stub Syntax - Extended Entry 3-20
Decision Table Entry Syntax - Extended Entry 3-20
Decision Table Initial Action - Extended Entry 3-20
Decision Table Verbs and Operations. 3-21
Introduction 3-21
Comparison Operators 3-21
One-of-a-set 3-21
Arithmetic Operations 3-22
Spacefill and Zeroise Operations 3-23
Fill Operation 3-23
Logical Operations 3-24
AND, OR and EXCLUSIVE OR 3-24
Logical AND 3-24
Logical OR 3-25
Logical EXCLUSIVE OR 3-25
SHIFT LEFT and SHIFT RIGHT 3-25
Logical Shifts 3-25
Move Operation MV 3-26
Moves of Similar Field Types 3-26
Character Moves 3-26
Binary Moves 3-26
Moves of Dissimilar Field Types 3-27
Character to Binary Moves 3-27
Binary to Character Moves 3-27
Hexadecimal Literal to Character Moves 3-27
Hexadecimal Constant to Binary 3-27
Multiple Move Operation MV 3-27
Decision Table Linking 3-28
Action Verb GOTO 3-28
Action Verb REPEAT 3-29
Action Verbs CALL, EXIT 3-29
Condition Verb IF, Action Verb EXIT 3-30
Action Verb IGNORE 3-31
Note on Linkage Diagram 3-31

Linkage Diagram 3-32
Action Verb DISPLAY. 3-33
Action Verb STOP 3-33
Other Action Verbs. 3-33
Decision Table and Fixed Tabulating Logic. 3-33
Decision Table Techniques. 3-34
Rule Mask Technique 3-34
Conversion of Decision Tables to Limited Entry Format 3-35
Decision Tables Summary 3-39
Decision Table Terminology 3-39
Worked Example. 3-39
Chapter 4 Fixed Logic and Printing. 4-1
Field Specifying Characters (FSCs) 4-1
*ALTER Directive. 4-2
Dummy Totalling Fields 4-4
Chaining Fields in the *INLIST 4-5
Decision Points 4-6
Introduction 4-6
Record Area 4-6
Work Area 4-7
*INLIST Area 4-8
Sort Slot 4-9
Decision Table Entry 4-9
Returning to the Fixed Logic 4-9
START Decision Point 4-9
RECORD Decision Point. 4-10
Use of RECLEN at the RECORD Decision Point. 4-11
BREAKH Decision Point 4-13
BREAK Decision Point 4-13
ENDFILE Decision Point 4-15
Closing The Main File Early 4-16
PRINT Decision Point 4-16
Tables Accessed by the Fixed Logic. 4-17
*LOOKUP Directives 4-18

*LOOKUP RECORD . . . 4-18
*LOOKUP BREAK x . . . 4-19
*LOOKUP PRINT . . . 4-20

Structure of the Fixed Logic . . . 4-20

Introduction . . . 4-20
Fixed Logic with *INLIST but no *SORT . . . 4-20
START DECISION POINT . . . 4-20
PRINT *HEADs . . . 4-21
RECORD DECISION POINT . . . 4-21
ENDFILE DECISION POINT . . . 4-22
END OF INPUT RECORD SET . . . 4-22
CONTROL BREAK TESTING . . . 4-23
BREAK DECISION POINT . . . 4-23
PRINT *OUTs . . . 4-23
ROLL TOTALS . . . 4-23
Note on Print Logic . . . 4-24
Fixed Logic with *INLIST and *SORT . . . 4-24
START DECISION POINT . . . 4-24
RECORD DECISION POINT . . . 4-24
INPUT TO THE SORT . . . 4-24
ENDFILE DECISION POINT . . . 4-25
END OF INPUT RECORD SET . . . 4-25
SORT . . . 4-25
PRINT *HEADs . . . 4-25
GET NEXT SORTED RECORD . . . 4-26
CONTROL BREAK TESTING . . . 4-26
BREAK DECISION POINT . . . 4-26
PRINT *OUTs . . . 4-27
ROLL TOTALS . . . 4-27
Note on Print Logic . . . 4-27
Diagram of Sort Record Layout for 'totals-only' *SORT . . . 4-27
Fixed Logic with No *INLIST . . . 4-28
START DECISION POINT . . . 4-28
RECORD DECISION POINT . . . 4-28
ENDFILE DECISION POINT . . . 4-29
END OF INPUT RECORD SET . . . 4-29

*TITLE Directive . . . 4-29

Format of a *TITLE Directive . . . 4-30

The 'b' Operand 4-30
The 'a' Operand 4-30
*TITLE Detail Lines 4-30
Content of *TITLE Detail Lines 4-31
*HEAD Directive 4-31
Format of a *HEAD Directive 4-31
The 'control-level(s)' Operand 4-31
Printing Order of Headings 4-32
Printing *HEAD L 4-32
Special Rules for *HEAD L with another Control Level Specified 4-33
Paper Control and Headings 4-33
The 'b' Operand 4-34
'Absolute' Positioning 4-34
'Relative' Positioning 4-34
Requesting New Pages at Certain Control Break Levels. 4-35
Special Rule for Headings 4-35
The 'a' Operand 4-36
The 'p' Operand 4-36
Content of *HEAD Detail Lines 4-37
Substitution Rules for Character Control and Transfer Fields. 4-37
Substitution Rules for Binary Control and Transfer Fields 4-38
Transfer Fields Printed in Headings 4-39
*OUT Directive. 4-39
Format of a *OUT Directive 4-39
The 'control-level(s)' Operand 4-39
Printing Order of Output Blocks 4-40
Paper Control and Output Blocks 4-40
The 'b' Operand 4-40
'Absolute' Positioning 4-40
'Relative' Positioning 4-40
The 'a' Operand 4-41
The 'p' Operand 4-41
Content of *OUT Detail Lines. 4-42
Literal Text and Special Strings in *OUT Detail Lines. 4-43
Control and Transfer Field FSC Strings 4-43
Totalling Field FSC Strings 4-43

*OPTION ZS - Leading Zero Suppression 4-43
*OPTION CZS - Complete Zero Suppression 4-43
*OPTION NZS - No Leading Zero Suppression 4-44
Editing Symbols within Numeric Fields 4-44
Floating Symbols 4-44
Editing Symbols 4-44
Examples of Editing and Floating Symbols 4-44
Post-signing of Totalling Fields 4-45
Printer Control Characters (Format Effectors) 4-45
Chapter 5 Program Control 5-1
Naming the Program 5-1
* OPTIONs 5-1
Compiler Listing *OPTIONs 5-2
Other Compiler *OPTIONs 5-3
Program Run-time *OPTIONs 5-3
Incorporating External Source 5-7
*COPY Directive 5-7
Terminating the Program 5-8
Formatting the Compilation Listing 5-8
Chapter 6 Files, Tables 6-1
The Main Input File 6-1
*FILE 6-1
Examples 6-3
LOOKUP Verb and *FILEs 6-3
Notes 6-3
Use of FF, VV and SUBCOUNT 6-3
Closing The Main File Early 6-5
Other Input File 6-5
*IFILE Directive 6-5
Examples 6-6
Examples 6-7
Examples 6-7
READ Verb and *IFILEs 6-7
LOOKUP Verb and *IFILEs 6-9

Closing *IFILEs. 6-9
Printer Output File. 6-9
Other Output Files. 6-10
*OFILE/*EFILE Directives. 6-10
Examples. 6-11
Examples. 6-12
WRITE Verb and *OFILEs/*EFILEs 6-12
Closing *OFILEs/*EFILEs 6-13
Input/Output Files 6-14
*IOFILE Directive. 6-14
Examples. 6-15
Examples. 6-16
Examples. 6-16
WRITE Verb and *IOFILEs 6-16
READ Verb and *IOFILEs 6-18
LOOKUP Verb and *IOFILEs 6-20
INDEX Verb and *IOFILEs 6-20
DELETE Verb and *IOFILEs 6-21
Closing *IOFILEs 6-21
Interaction of Verbs and File Organisation. 6-22
Serial Files (OR=L) 6-22
Indexed-Sequential Files (OR=I). 6-23
Direct Files (OR=D) 6-23
Accessing Environment Variables 6-24
Tables 6-24
*TABLE Directive. 6-24
READ Verb and *TABLEs 6-25
WRITE Verb and *TABLEs 6-26
LOOKUP Verb and *TABLEs 6-27
SELECT Verb and *TABLEs 6-28
DELETE Verb and *TABLEs 6-28
CLOSE Verb and *TABLEs 6-28
'Key-only' *TABLEs 6-29
Examples of *TABLE Access 6-29
Example 1: Successive READs 6-29

Example 2: READ Following 'true' LOOKUP . . . 6-29
Example 3: READ Following 'false' LOOKUP . . . 6-29
Example 4: READ Following SELECT . . . 6-30
Example 5: WRITE Following 'true' LOOKUP . . . 6-30
Example 6: WRITE Following 'false' LOOKUP . . . 6-30
Example 7: WRITE Following SELECT . . . 6-30
Example 8: DELETE Following 'true' LOOKUP . . . 6-30
Example 9: DELETE Following 'false' LOOKUP . . . 6-31
Example 10: DELETE Following READ . . . 6-31
Worked Example . . . 6-31
Chapter 7 Reserved Words, Restricted Names . . . 7-1
Reserved Words . . . 7-1
Use of BREAK, LEVEL and PP-3 . . . 7-3
Use of FF, VV and SUBCOUNT . . . 7-4
Use of COUNT . . . 7-4
Use of RECLEN . . . 7-5
Use of LINES . . . 7-5
Use of PP0 . . . 7-5
Access to Reserved Words . . . 7-6
Restricted Names . . . 7-7
Restricted Decision Table Names . . . 7-7
Chapter 8 Field Definition and Pointers . . . 8-1
Alternative Field Definition Formats . . . 8-1
Restrictions on Field Start-positions and Lengths . . . 8-4
Introduction to Pointers . . . 8-5
Pointer Definitions . . . 8-6
Initialising Pointers . . . 8-6
Updating Pointers . . . 8-8
Pointer Fields . . . 8-9
Examples . . . 8-10
Operations using Pointer Fields . . . 8-10
Example of an Application using Pointers . . . 8-10
Worked Example . . . 8-11
Pointer Skip Operators . . . 8-13

Chapter 9 Techniques 9-1
Calculation and Field Manipulation 9-1
Multiplying a Binary Field by a Power of 2 9-1
Arithmetic on Record Area Fields 9-1
Field Types. 9-2
Operations on Character Strings 9-2
Calculation to a Fixed Number of Decimal Places 9-3
Decision Tables 9-3
Structure of Programs 9-3
Conditions 9-4
ELSE Rule 9-4
Actions 9-4
Extended Entry Format 9-5
Conditions in Extended Entry 9-5
Identical Entries in Extended Entry Decision Tables 9-6
Decision Points 9-7
Pointers 9-7
Alignment 9-8
Binary Fields 9-8
Character Fields 9-9
Pointer Fields 9-9
Internal Format 9-9
Numeric Constants 9-9
Decision Tables 9-9
Grouping of Fields 9-10
*INLIST 9-10
Headings and Print Lines 9-10
Chapter 10 External Routines 10-1
EXECUTE Verb 10-1
Example 10-1
Chapter 11 Examples 11-1
Simple Listing 11-1
Listing with Final Totals 11-2

Listing with Control Break Totals . . . 11-3
Listing with Selection and Flagging . . . 11-4
Processing at the RECORD Decision Point . . . 11-6
Processing at the BREAK Decision Point . . . 11-7
File Amendment . . . 11-9
File Matching . . . 11-10
Updating an Indexed-sequential File . . . 11-11
File Merging . . . 11-13
File Updating . . . 11-14
Data Input and Validation . . . 11-15
Totalling Using *TABLE . . . 11-18
Totalling using Arrays and Pointers . . . 11-19
N-up Printing - Different Items . . . 11-21
Enclosing Negative Values in Brackets . . . 11-22
Reading an I-S File using an Alternate Index . . . 11-24
Updating an I-S File using an Alternate Index . . . 11-25

Appendix A Operations Guide . . . A-1

Overview . . . A-1
- What Your Operations Guide Contains . . . A-1

UNIX and Linux Operations Guide . . . A-2
- Installation . . . A-2
  - RPM Installation . . . A-2
  - Unzip the tar File (release_name.tar.gz) . . . A-3
  - Retrieve the Installation Script . . . A-3
  - Run the Installation Script . . . A-3
  - Linking FILETAB into the /usr Hierarchy . . . A-3
  - Reconfiguring FILETAB . . . A-4
- Additional Steps Before Using FILETAB . . . A-4
- To Remove FILETAB from Your System . . . A-4
- Directory Structure and Environment Variables . . . A-4
  - Executables . . . A-4
    - FILETAB Compiler . . . A-5
    - ftc Shell Script . . . A-6
    - Sample Scripts . . . A-7

Libraries A-7
Windows NT Operations Guide A-8
Installation A-8
SFX Installation A-8
InstallShield Installation A-8
Additional Steps Before Using FILETAB A-8
To Remove FILETAB from Your System A-9
Directory Structure and Environment Variables A-9
ftc Batch File A-9
ftcomp Executable A-9
Libraries A-10
Include Files A-10
The Configuration File A-11
FILETAB Options File A-11
The Macros File A-11
Type Definition and Portability Files A-11
Run-time Support Function Prototypes A-12
Standard C (Library) Include Files A-12
Compiling Programs A-12
Compiler Output A-12
Running Programs A-12
Input and Output Files A-12
Temporary Sort Files A-13
Result Codes A-14
Compile-time Result Codes A-14
Run-time codes A-14
Appendix B Additional Field Types: Dates, Strings, Floating Points B-1
Date Fields B-1
Field Definition B-1
Operations and Date Fields B-1
Conversion To and From Date Fields B-1
Date Literals B-2
Associating Date Fields with FSCs in the *INLIST (or a *LIST) B-2
Printing of FSCs Associated with Date Fields B-2
*OPTION DF B-3
*OPTION DW B-4

SPLITDATE Verb B-4

String Fields B-5

Field Definition B-5

Treatment of String Fields B-5

Floating Point Fields B-6

Field Definition B-6

Operations and Floating Point Fields B-6

Conversion To and From Floating Point Fields B-6

Numeric Constants B-7

Associating Floating Point Fields with FSCs in the *INLIST (or a *LIST) B-7

Printing of FSCs Associated with Floating Point Fields B-7

*OPTION DP B-7

Other Types of Field B-7

Appendix C Miscellaneous Features: Conditional Compilation C-1

Conditional Compilation (*COND) C-1

*COND DEFINE Flag C-1

*COND IF Flag C-1

*COND IFNOT Flag C-1

*COND ELSE C-1

*COND ENDIF C-1

*COND ENDALL C-2

Appendix D FILETAB on the Web D-1

Overview D-1

What is FILETAB on the Web? D-1

Filetab on the Web Features D-1

What Your Operations Guide Contains D-1

Adding the Web Features D-2

The Original Program: Simple Text Output D-3

Add HTML Output D-5

Add an Index Page D-6

How to Add Indexing D-8

Add a Company Logo or Graphics to the Output D-9

Run the Program Using HTML Form Input D-11

Form Input: How It’s Done D-12

The Program Altered for CGI Use: test6.ftc D-13
Run the Program Using a Hypertext Link D-15
More About the New *OPTIONs and FTC_HTML D-17
Switch on HTML Output at Run-time D-17
Switch on HTML Output at Compile-time D-17
Summary of Compile-time and Run-time Options D-18
More About *MAPs D-19
Using *MAP Detail Lines to Read from CGI Forms D-19
File Organisation OR=CGI D-19
MAP Format for CGI D-19
MAP Directive Examples D-20
MAP Detail Examples D-21
Doing the READ. D-21
Using *MAP Detail Lines to Create Better Input Forms D-22
Summary D-22
Examples D-23
Listing of mapdetail.ftc D-25
Appendix E Informix C-ISAM E-1
Overview E-1
Organisation E-2
Record Length E-2
Keys E-2
Record Locking E-4
Transaction Locking E-5
Verbs and C-ISAM Files E-6
COMMIT File E-6
ROLLBACK File E-6
RELEASE File E-6
INDEX File n E-6
Reserved Words and C-ISAM Files E-7
ZREPLY E-7
Field Types Supported With C-ISAM E-8
Internal Decimal Fields E-9
Field Definition E-9

Operations and Internal Decimal Fields E-9
Conversion To and From Internal Decimal Fields E-9
Numeric Constants. E-10
Associating Internal Decimal Fields with FSCs in the *INLIST (or a *LIST) E-10
Printing of FSCs associated with Internal Decimal Fields E-10
External Decimal Fields E-10
Field Definition E-10
Treatment of External Decimal Fields E-11
Appendix F ODBC Guide F-1
What Your ODBC Guide Contains F-1
Keywords and ODBC Files F-1
Organisation F-2
SQL Directive Name F-2
Record Length F-2
INDICATORS Directive Name F-2
Error Handling *DETAB F-2
Database F-2
User F-3
Row Cache F-3
Directives and ODBC Files F-3
*SQL Directive F-3
*INDICATORS Directive F-4
Verbs and ODBC Files F-4
CONNECT file_name F-4
DISCONNECT file_name F-4
CLOSE file_name F-4
READ file_name Field F-4
EXECSQL file_name sql_directive_name F-4
Reserved Words and ODBC Files F-5
Field Types Supported with ODBC F-5
Examples F-5
Example 1 Sales Analysis F-6
Example 2 Despatch Note F-8

Appendix G Informix Guide G-1

About your Informix Guide G-1

Keywords and Informix Files G-1

Organisation G-2

SQL Directive Name G-2

Record Length G-2

INDICATORS Directive Name G-3

Error Handling *DETAB G-3

Database G-3

Directives and Informix Files G-3

*SQL Directive G-3

*INDICATORS Directive G-4

Verbs and Informix Files G-4

CONNECT file_name G-4

DISCONNECT file_name G-4

CLOSE file_name G-5

READ file_name Field G-5

EXECSQL file_name sql_directive_name G-5

Reserved Words and Informix Files G-5

Field Types Supported with Informix G-6

How FILETAB Prints Informix NULL Fields G-7

Internal Decimal Fields G-7

Examples G-7

Example 1 . . . . G-8

Example 2 . . . . G-9

Appendix H INGRES Guide H-1

About your INGRES Documentation H-1

Keywords and INGRES Files H-1

Organisation H-2

SQL Directive Name H-2

Record Length H-2

INDICATORS Directive Name H-3

Error Handling *DETAB H-3

Database H-3
Directives and INGRES Files H-3
*SQL Directive H-3
*INDICATORS Directive H-4
Verbs and INGRES Files H-4
CONNECT file_name H-4
DISCONNECT file_name H-4
CLOSE file_name H-4
READ file_name Field H-5
EXECSQL file_name sql_directive_name H-5
Reserved Words and INGRES Files H-5
Field Types Supported with INGRES H-6
How FILETAB prints INGRES NULL fields H-6
Examples H-6
Example 1 . . . . H-7
Example 2 . . . . H-8
Example 3 . . . . H-9
Appendix I ORACLE Guide . . . . I -1
About your ORACLE Guide . . . . I -1
Keywords and ORACLE Files . . . . I -1
Organisation . . . . I -2
SQL Directive Name . . . . I -2
Record Length . . . . I -2
INDICATORS Directive Name . . . . I -3
Error Handling *DETAB . . . . I -3
Database . . . . I -3
User . . . . I -3
Directives and ORACLE Files . . . . I -3
*SQL Directive . . . . I -3
*INDICATORS Directive . . . . I -4
Verbs and ORACLE Files . . . . I -4
CONNECT file_name . . . . I -4
DISCONNECT file_name . . . . I -4

CLOSE file_name . . . . I -5
READ file_name Field . . . . I -5
EXECSQL file_name sql_directive_name . . . . I -5
Reserved Words and ORACLE Files . . . . I -5
Field Types Supported with ORACLE . . . . I -6
How FILETAB Points ORACLE NULL Fields . . . . I -6
Examples . . . . I -6
Example 1 . . . . I -7
Example 2 . . . . I -8
Example 3 . . . . I -9
Appendix J HTMLX Guide . . . . J -1
Overview . . . . J -1
About Your HTMLX Documentation . . . . J -1
Options and HTMLX . . . . J -1
Reserved Words and HTMLX . . . . J -2
HTML Decision Point and HTMLX . . . . J -3
Open Hook . . . . J -3
Exit Options . . . . J -3
Amending the HTML Data String . . . . J -4
Close Hook . . . . J -4
Exit Options . . . . J -4
Amending the HTML Data String . . . . J -4
Print hook : Zone 0 - Start of Table Row. . . . . J -4
Exit Options . . . . J -5
Amending the HTML Data String . . . . J -5
Print hook : Zone n - Table Cell . . . . J -5
Exit Options . . . . J -6
Amending the HTML Data String . . . . J -6
Zero Hook . . . . J -6
Mapline and Zone Information Generation. . . . . J -6
Examples . . . . J -6
Example 1 Static HTML Output . . . . J -7
Example 2 Dynamic HTML Output . . . . J -8

Chapter 1
Introduction

1.1 About FILETAB

FILETAB is an easy-to-use yet powerful and flexible programming tool, developed for commercial and business applications, it is available for a wide range of computers. The aims of FILETAB are to enable computer departments to write and develop programs quickly, and to produce programs which are easily maintained and modified. FILETAB is also easy to learn, and a subset can be taught to non-IT specialists.

The sorting routines incorporated in FILETAB enable complex reports to be produced by a single program. Run-times are reduced significantly when, for example, record selection and sorting take place in the same program.

1.1.1 Programming

FILETAB programs are simple to write and understand. This is largely because all program logic is specified using decision tables.

FILETAB may be used for writing a variety of commercial programs. Features of the software enable data input, file creation, updating, merging and report production to be performed efficiently. FILETAB may also be used as a general utility, especially with its sorting and tabulation facilities.

FILETAB combines all the facilities expected of a report generator, and several more besides.

1.1.2 Report Generation

Specifying a few lines of code enables the user to read a file automatically, format and print data and headings, accumulate totals and check for control breaks. A few more lines allow the reading of subsidiary files, selection of data, calculation, and sorting of the report.

FILETAB is ideally suited to report production, where the components of the output are often already known, but an acceptable format may not have been determined. In FILETAB, printed output is represented by ’picture images’, and print layouts can be amended rapidly without any other changes to a program. When using FILETAB, it is perfectly reasonable for a program to be tested with different output specifications until the ideal report format is found.

FILETAB’s facilities for writing reports can be learned in a few days. Users need to understand only the basic features of FILETAB, since the Fixed Logic provides an ideal foundation for report programs; non-IT specialists are frequent users of FILETAB. Decision tables are used to identify records of special interest, and the in-built sort and tabulator are used for producing the reports.

1.2 About your FILETAB Documentation

Each chapter in the FILETAB Developers’ Guide is divided into logical sections. For quick reference, an overview of each chapter follows, to search for a specific subject use the comprehensive contents list.

Chapter 1 Introduction	Discusses the main features of FILETAB and the documentation
Chapter 2 Simple Reports	Explains the FILETAB Fixed Logic in detail and describes the formats of the directives used in the production of reports from a single file
Chapter 3 Decision Tables	All program logic, processing, record selection etc. in FILETAB must be specified in decision tables. The first section of this chapter introduces decision table theory, and the remainder describes the use of decision tables in FILETAB
Chapter 4 Fixed Logic and Printing	Describes the following: Field specifying characters, FILETAB Fixed Logic, FILETAB's internal storage areas, Decision Points, Tables accessed by the Fixed Logic, Directives that control printing and Printer control characters (Format Effectors)
Chapter 5 Program Control	Describes the following: naming the program, compiler listing control (compiler listing `*OPTION`s), other complier `*OPTION`s, program run-time `*OPTION`s, incorporation of external source, termination of the program and formatting the compilation listing
Chapter 6 Files, Tables	Describes how: more than one input file may be accessed, output files may be accessed, Indexed-Sequential and Direct files may be accessed and updated lookup tables may be formed and accessed
Chapter 7 Reserved Words, Restricted Names	A number of fields have been allocated to help the programmer access special data fields used by the Fixed Logic. The names given to these fields are reserved and must not be defined in the `*DICTIONARY`. This chapter describes these 'Reserved Words'

Chapter 8 **Field Definition and Pointers**	A 'Field Definition' describes the field's position, type and length. Field Definitions are normally associated with 'Fieldnames' by entries in the `*DICTIONARY`. A 'Pointer' is a data item which is used to perform indirect addressing and string-handling in FILETAB programs. Pointers contain address variables when they are referenced within decision tables
Chapter 9 **Techniques**	This chapter deals in some detail with programming style, and with the techniques that lead to more efficient use of FILETAB
Chapter 10 **External Routines**	It may be appropriate or convenient to incorporate within FILETAB programs, routines written in other programming languages (such as 'C'). The mechanism within FILETAB used to facilitate this is the `EXECUTE` verb
Chapter 11 **Examples**	Demonstrating the uses of FILETAB
Appendix A **Operations Guide**	Provides instructions for installing the FILETAB system software and making it available to developers. It also describes how to compile and run your programs
Appendix B **Additional Field Types: Dates, Strings, Floating Points**	Describes the use of additional field types within FILETAB
Appendix C **Miscellaneous Features: Conditional Compilation**	Describes the use of conditional compilation and its use for tailoring programs whilst maintaining a single source
Appendix D **FILETAB on the Web**	Describes the FILETAB extensions, introduced in Version 3.0, that allow you to access your company data on web browsers
Appendix E **Informix C-ISAM**	Describes the keywords, verbs and reserved words required to access Informix C-ISAM index sequential files
Appendix F **ODBC Guide**	Provides comprehensive details of how to access data held in an ODBC compliant database
Appendix G **Informix Guide**	Provides comprehensive details of how to access data held in an Informix compliant database
Appendix H **INGRES Guide**	Provides comprehensive details of how to access data held in an INGRES database

Appendix I ORACLE Guide	Provides comprehensive details of how to access data held in an ORACLE database
Appendix J HTMLX Guide	HTMLX allows FILETAB to produce enhanced web pages such as: automatic HTML table based versions of FILETAB output utilising fonts, colours, formats, page links and so on; non-table based pages such as graphic reports, data based forms; XML/XSL output and sophisticated Javascript (or VBScript) applications such as report outliners, complex tree navigators

1.3 Compiling and Running FILETAB Programs

1.3.1 On UNIX and Linux

To compile and run a FILETAB program in the file `program_name.ftc` use the script `ftc` as follows:

```
ftc -g program_name
```

You may need to include the directory where `ftc` was installed in your `PATH`.

For details on compiling and running FILETAB programs separately and further details on configuring the FILETAB environment. *(See Appendix A Operations Guide)*.

1.3.2 To Compile a FILETAB for Windows NT Program

To compile a FILETAB program in the file `program_name.ftc` use the batch command **`ftc`** as follows:

```
ftc program_name
```

If successful, this will create an executable file `program_name.exe`

You may need to include the directory where FILETAB was installed in your `PATH` statement.

1.3.3 To Run a Compiled FILETAB for Windows NT Program

To run the program compiled in 1.2.2, type the name of the executable file as follows:

```
program_name
```

1.4 Principles

1.4.1 Source Statements

FILETAB is controlled by source statements of which there are three types:

- Directives
- Detail statements
- Comments

Directives have an `*` in position 1, followed by the name of the directive. Only the first three characters are used to identify the directive, so abbreviations can be used. Throughout this manual, the full form of all directives is used for clarity. A directive may be followed by detail statements, which provide more information. A series of detail statements is terminated by the next directive, unless this is `*PAGE` or `*COPY`. On some directives, additional fields known as 'operands' may be specified, and these too provide extra information. Some directives and operands may be omitted, in which case default values are assumed.

Most directives cause computer instructions to be generated. Comments, however, do not produce computer instructions, but are used simply as a program documentation aid.

All directives are described in detail in the following chapters.

1.4.2 Fixed Logic

Many code features required when using languages such as FORTRAN and COBOL (for example, totalling, checking for control breaks and so on) are handled automatically by FILETAB, through processing known as the Fixed Logic. In order to obtain maximum benefit from using FILETAB it is important that Fixed Logic is properly understood. *The Fixed Logic is described in detail in Chapter 4, but a brief summary follows.*

Using `*FILE` and `*DICTIONARY,` the file from which the report is to be produced (known as the main file) must be described. Fields to be totalled, fields causing control breaks and other fields to be printed are defined using the `*INLIST` directive. Report titles, headings and output requirements are defined by means of images of the output required, which are supplied as `*TITLE`, `*HEAD` and `*OUT` directives. Program logic and processing is defined by means of `*DETAB` (decision table) directives.

FILETAB automatically reads each record from the main file into an area of store known as the Record Area.

FILETAB then:

- Processes each record according to the logic specified by the user in `*DETAB`s

- Totals the appropriate fields
- Prints the record in the defined format
- Outputs the control break totals as required
- Prints the headings at the appropriate points

As mentioned previously, the record from the input file is read into an area known as the Record Area, where it may be accessed from decision tables. The remainder of the Record Area, or an area known as the Work Area, may also be accessed from decision tables for use as a general purpose storage and processing area. These are described in detail in *Record and Work Areas on page 12 of Chapter 2 and Restrictions on Field Start-positions and Lengths on page 4 of Chapter 8.*

1.5 Example Files

Most of the examples contained in this document refer to two files: a customer master, and a transaction file. The formats of records in customer master follow:

LOCALNAME: CUSTFILE **ORGANISATION: SERIAL** **RECORD LENGTH: 332 BYTES**				
Customer Master File				
Position (bytes)				**Description**
From	**To**	**Length**	**Data Type**	
0	1	2	Char	Record type = 05
2	3	2	Char	Area - Numeric
4	7	4	Char	Account number - Numeric
8	43	36	Char	Account name
44	147	104	Char	Account address - four lines 32, 32, 32, eight bytes long
148	159	12	Char	Account telephone number
160	195	36	Char	Statement name
196	299	104	Char	Statement address - four lines 32, 32, 32, eight bytes long
300	309	10	Char	Sort key
310	310	1	Char	Closed indicator
311	316	6	Char	Date of creation - DDMMYY
317	319	3	Char	Filler
320	323	4	Binary	Statement count
324	327	4	Binary	Outstanding balance - binary pence
328	329	2	Char	Credit code - Numeric
330	331	2	Char	Customer type

Table 1-1: Custom Master

The formats of records in a transaction file follows:

LOCALNAME: TRANSDY **ORGANISATION: SERIAL** **RECORD LENGTH: 64 BYTES**				
Transaction File				
Position (bytes)				**Description**
From	**To**	**Length**	**Data Type**	
0	1	2	Char	Transaction type = 21
2	3	2	Char	Area - Numeric
4	7	4	Char	Account number - Numeric
8	13	6	Char	Invoice number - Numeric
14	19	6	Char	Reference - Alphanumeric
20	25	6	Char	Date - DDMMYY
26	27	2	Char	Filler
28	31	4	Binary	Value - pence
32	35	4	Binary	VAT - pence
36	39	4	Binary	Discount - pence
40	51	12	Char	Order no. - Alphanumeric
52	54	3	Char	Product code - Numeric
55	57	3	Char	Quantity code - Numeric
58	59	2	Char	Filler
60	63	4	Binary	Quantity - Numeric

Table 1-2: Transaction File

1.6 Conventions Used in this Guide

The following typographical conventions are used throughout this document:

Typeface	Used For
`Courier`	System messages, file, path and directory names
Italic	Notes and referencing titles of documents
Bold	Emphasis, system buttons, options and settings
CAPITALS	Acronyms and abbreviations
`Courier Bold`	Commands and keyboard options

Chapter 2
Simple Reports

This chapter explains the FILETAB Fixed Logic in detail and describes the formats of the directives used in the production of reports from a single file.

2.1 Preliminary Notes

Simple report programs may logically be split into two. The first part specifies the location and format of the input data, and the second specifies the format and content of the report required.

The simplest possible application is to print data from a file. In the following example, the contents of the file are held in character format.

Localname: STOCK		
Position (bytes)		Contents
From	To	
0	4	Product Group
5	20	Part Number
21	40	Description of Part
41	45	Quantity
46	53	Price

Table 2-1: File contents in character format

To print this file, FILETAB needs to know where to find the data, and which items of it to print. In this example it is all to be printed.

To identify the location of the data, the `*FILE` directive is used, together with the 'localname', which links the program with the physical file named using the `NAME` keyword:

```
*FILE STOCK  NAME=stock.txt
```

Records from the file specified here are placed automatically into the Record Area, from byte zero onwards.

Items which are to be printed are moved automatically into the `*INLIST` Area, and to reserve space for them, the `*INLIST` directive is used. Space is reserved by associating a character (known as a field specifying character) with the fields to be printed:

```
*INLIST
A  0/54
```

Here, `A` is linked with a description of the area occupied by the input data in the Record Area. 0 represents the first byte of data, `/` indicates that the data is in character format, and `54` indicates its length.

FILETAB now has all the necessary information to print the data. The directive `*GO` completes the program.

At this point it would be usual to give the program a name. The `*PROGRAM` directive (see Chapter 2.3) is used for this purpose:

```
*PROGRAM EXAMPLE1
```

The complete program looks like this:

```
*PROGRAM EXAMPLE1
*FILE STOCK  NAME=stock.txt
*INLIST
A 0/54
*GO
```

A sample line of the output appears below:

```
0912131-360-2048-0015C/SUNK BOLTS  3MMX20 4200     350
```

It was stated above that simple report programs may logically be divided in two. The example above deals with the first part of the program (specification of the location and format of the input data); the `*INLIST` acts as a bridge, linking it with the second part, the content and format of the output, which is left to FILETAB, and is handled automatically.

In the next example, the program incorporates three refinements:

(1) It selects particular items of data for printing.

(2) It specifies the format in which they are to be printed.

(3) It uses an editing technique to include a decimal point in 'Price'. The file used is the same file as for the first example - 'STOCK'.

The location of the required data is identified as before:

```
*FILE STOCK  NAME=stock.txt
```

Items to be printed are moved automatically into the 'Inlist Area' by FILETAB, again by way of the `*INLIST` directive, but this time only three items are selected: Part Number, Description and Price. To keep the items separate, and to allow for formatting and editing, they are associated with three different 'field specifying characters' or FSCs, thus:

```
*INLIST
A  5/16
B  21/20
C  46/8
```

Here again, the number to the left of the oblique indicates the start-position of the data, and the number to the right indicates its length.

To specify the format of the output, a picture-image is created, using the FSCs associated with the data items. This image is introduced by the `*OUT` directive:

```
*OUT  L
  AAAAAAAAAAAAAAAA     BBBBBBBBBBBBBBBBBBBB    CCCCCC.CC
```

The L on the *OUT directive stands for List and indicates that this line is to be printed for every record. Further details may be found in *Chapter 2.9*.

To complete the program, the *GO directive is used, and the program is named. The program now looks like this:

```
*PROGRAM EXAMPLE2
*FILE STOCK  NAME=stock.txt
*INLIST
A  5/16
B  21/20
C  46/8
*OUT  L
  AAAAAAAAAAAAAAAA     BBBBBBBBBBBBBBBBBBBB    CCCCCC.CC
*GO
```

A sample line of output follows:

```
  31-360-2048-0015    C/SUNK BOLTS 3MMX20           3.50
```

Note: As there is a decimal point within the string of characters that represents the 'Price' field, a decimal point has been placed in the corresponding position in the output. All such editing features are fully described in Chapter 4.

So far, we have considered the use of four directives: *FILE, *INLIST, *OUT and *GO. A further, optional, but very important, directive is *DICTIONARY. This allows specific data fields to be named, and fields described by these names in subsequent processing.

For instance, if a *DICTIONARY had been included in the last program, it would look like the following example.

```
*PROGRAM EXAMPLE2
*FILE STOCK  NAME=stock.txt
*DICTIONARY
PARTNO    = 5/16
PARTDESC  = 21/20
PRICE     = 46/8
*INLIST
A  PARTNO
B  PARTDESC
C  PRICE
*OUT  L
  AAAAAAAAAAAAAAAA     BBBBBBBBBBBBBBBBBBBB    CCCCCC.CC
*GO
```

In this example, the fields that are to be used in the output are named in the *DICTIONARY, and these names are associated with FSCs in the *INLIST. These *DICTIONARY names may, of course, be used elsewhere in a program.

A major advantage of the use of a *DICTIONARY is that, should the format of records be altered, only the definition of fields in the *DICTIONARY need to be changed. The names associated with them, and used throughout the program, may be retained to simplify maintenance. The use of *DICTIONARY is strongly recommended, both for this reason and because names are more easily understood than the

'field definitions' that were used initially. 'PRICE' is more immediately meaningful than the corresponding field definition 46/8.

*Of the directives noted so far, *PROGRAM, *FILE, *DICTIONARY, *INLIST, *OUT and *GO, together with three further directives - *HEAD, *TITLE and *SORT - are described in detail later in this chapter, and in Chapter 4. Reference is also made to Decision Tables (*DETABs), which are described in detail in Chapter 3; to Record, Work and Print Areas; and to FILETAB's Fixed Logic, an automatic sequence of operations designed to assist in the production of report programs. The Record, Work and Print Areas, and the Fixed Logic are described in Chapter 4. First, however, certain further items must be described.*

2.1.1 Sequence of Directives

The directives mentioned in this chapter would normally appear in a program in the following order:

```
*PROGRAM
*FILE
*DICTIONARY
*INLIST
*TITLE
*HEAD
*OUT
*SORT
*GO
```

2.1.2 Operands

Some directives are followed by operands, which give more information about the directive,

- In these cases, operands should be separated from the directive by at least one space
- Spaces must not appear within an operand
- Multiple operands are separated by commas only.

The operands which are associated with each directive are described in detail in this and subsequent chapters, with examples of their use.

2.1.3 Detail Statements (Detail Lines)

The function of detail statements (detail lines) should become apparent when individual directives are discussed. For now, an example should suffice. The previous program example contains an *INLIST directive. The subsequent lines, which associate field specifying characters with names, are detail lines.

2.1.4 Comments

Comments may be included in FILETAB programs. A comment may appear on a line by itself, thus:

```
* comment
```

where the asterisk must occupy position 1, and a space must occupy position 2. Alternatively, a comment may be added to a detail statement, thus:

```
statement [comment
```

Comments must not appear on, or immediately after, detail statements associated with print-line images, that is, detail statements for `*TITLE`, `*HEAD` and `*OUT`.

2.1.5 Length of Lines

FILETAB has two modes of operation concerning command line length.

```
*OPT NCL
```

The `NCL` (No Continuation Lines) command can be included anywhere in the code, but must appear prior to code containing lines exceeding 76 characters. In this mode each `*TITLE`, `*HEAD` or `*OUT` directive line results in a SINGLE print line.

As code may be imported from other environments another mode concerning line lengths is available. The mode is specified by:

```
*OPT CL
```

The `CL` (Continuation Lines) command can be also be included anywhere in the code. It is assumed as the default mode. Within `CL` mode, the following rules apply:

(1) If any program lines (except `*TITLE`, `*HEAD`, `*OUT`, `*LOOKUP`, `*TABLE` detail lines and comment lines) contain non-space characters in columns 77-80 and only space characters after column 80, the following line is treated as a continuation line.

(2) `*LOOKUP`, `*TABLE` and comment lines cannot have continuation lines.

(3) `*TITLE`, `*HEAD`, `*OUT` detail lines cannot be longer than 80 characters. However, as the default print line length is 160 characters, every second detail line is therefore linked to the previous one to form each print line (whole detail lines are concatenated to form a print line of the required length).

The size of the print line can be defined or altered by the following command:

```
*OPT PL=integer
```

The integer is the line length in bytes. If a code line is shorter than the specified `PL` length (that is, the `PL` length is not a multiple of 80) then the remainder of the line is space filled.

If the print line length were re-set to 240 characters then every 3 detail lines (`*TITLE`, `*HEAD`, `*OUT`) would be concatenated to form each print line.

2.2 Fixed Logic

In FILETAB, the Fixed Logic provides the underlying framework upon which programs are built. The function of FILETAB directives is largely to inform and direct the Fixed Logic.

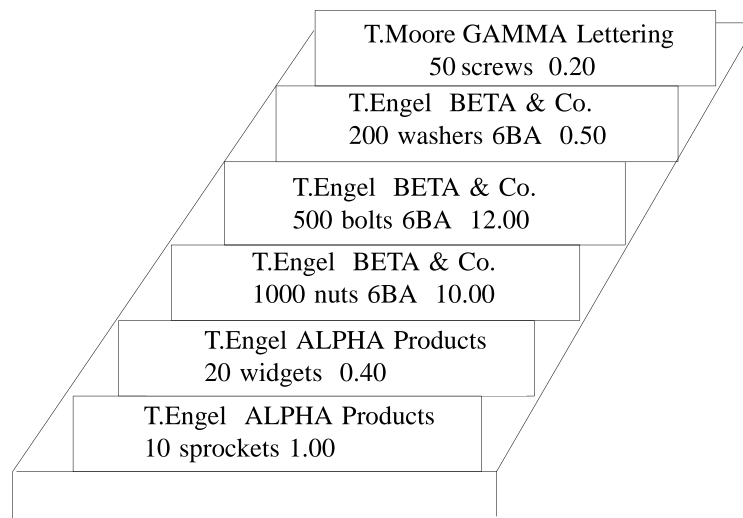
Using a manual clerical application as an example, the following describes how FILETAB Fixed Logic works.

The report below is part of a monthly sales report produced by the sales manager of a small company. The procedures are related throughout to some of the relevant operations of the Fixed Logic.

```
Monthly Sales Report                                    03/01/01
T.Engel  North West Sales

ALPHA Products

10 sprockets                          £1.00
20 widgets                            £0.40

Total for ALPHA Products              £1.40

BETA & Co.

1000 nuts 6BA                        £10.00
500 bolts 6BA                        £12.00
200 washers 6BA                       £0.50

Total for BETA & Co.                 £22.50

Total for T.Engel                    £23.90

T.Moore  North East Sales

GAMMA Lettering

50 screws                             £0.20

Total Sales for January            £1234.40
```

In the above report, the details of each transaction are grouped together under the name of the customer, and the names of customers are grouped together under that of the salesperson.

The sales manager uses a card index system, which contains details of every item sold, held in customer within salesperson order. A sample of the relevant index follows:



To prepare a report from this card index, the manager proceeds as follows:

(1) Write down the title of the report - 'Monthly Sales Report'. This is done in FILETAB by using the `*TITLE` directive.

(2) Obtain the first card from the index, and identify the salesperson, whose name is entered as a heading. On the next line, write down the name of the customer. These two pieces of information control the format of the report, and are known in FILETAB as 'control fields'. In this case, the salesperson's name is the major control field, and the customer's name is the minor control field. FILETAB caters for six such 'control levels'.

(3) Write down the details of the order from the first card. This is known as a 'detail line' in the report. One such line should exist for each line in the card index. This is known in FILETAB as the '`LIST` level', and the output line produced is a `*OUT L` line (`L` representing `LIST`).

(4) Take the next card, and checks that it refers to the same salesperson as the previous card. This is known in FILETAB as a 'control break test'. In this case, it does refer to the same salesperson, so check it again to see if it refers to the same customer as the previous card. In FILETAB terms, a control break test has been conducted, at the major (higher) level of control, and then at the minor level. Since neither control level has 'broken', simply enters details of the transaction under the previous one.

(5) Proceed to the next card, which contains a different customer's name. This is known in FILETAB as a 'control break'. There are now two things to do:

(a) Total the transactions for the previous customer, and add this total to the report. In FILETAB, this line is known as an *OUT line at this 'break level'

(b) Write down the name of the next customer as a heading and proceed to list the details for this customer

The sixth card pulled from the index is for a different salesperson. This is again a control break, but this time at a major control level. Just as when the customer's name changed, the manager must calculate totals, but this time two totals are required: the total for the transactions of that customer, and the total for the salesperson. These totals are followed by the heading for the new salesperson, and the heading for his first customer.

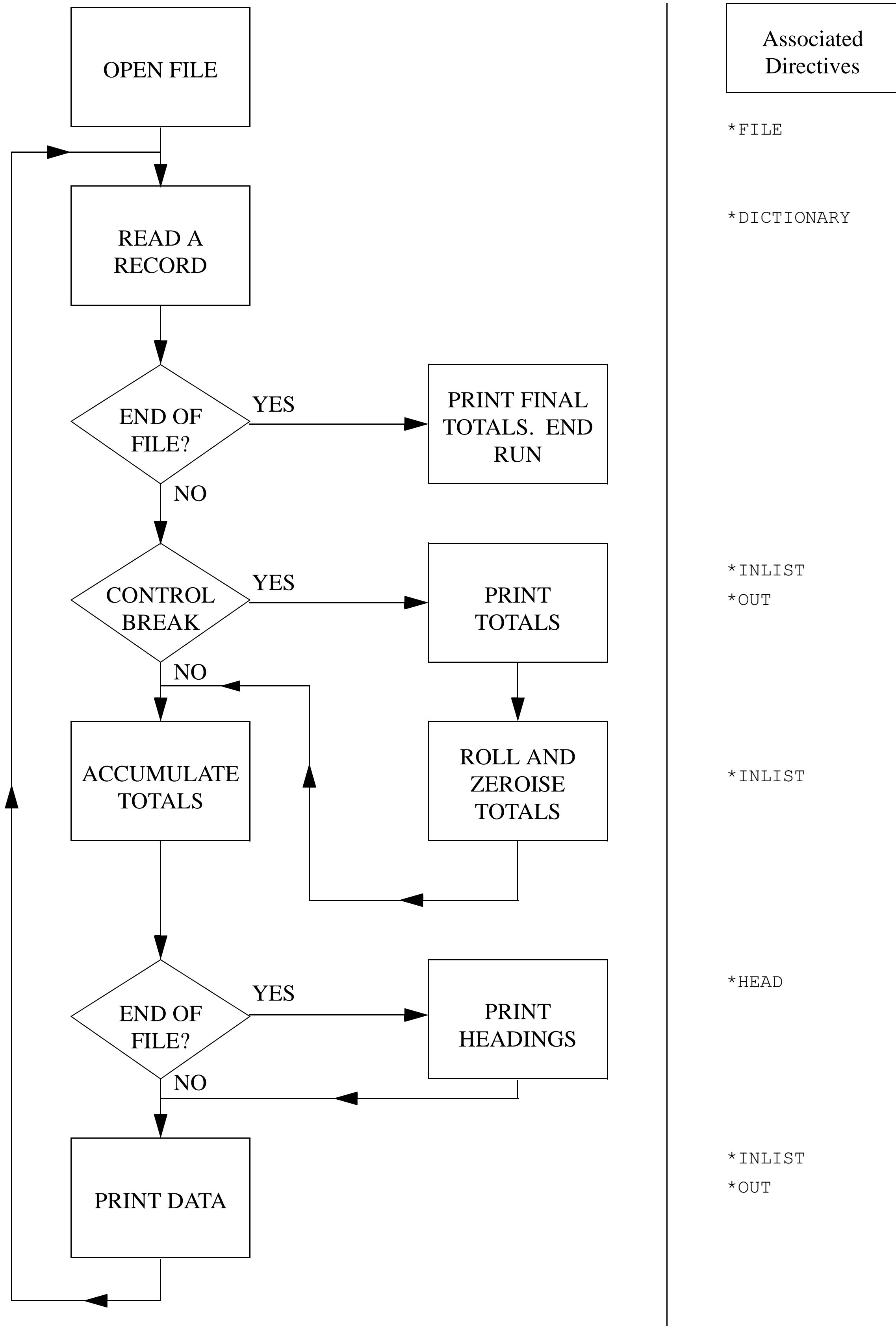
(6) The manager would probably calculate totals for salespeople by adding together the individual totals for customers; FILETAB calculates its totals by a variation of this method, in that each time a control break occurs, the total for that level of control is added into a running total for the next higher level. The total for a particular control break is then printed and cleared. This is known as 'rolling totals'.

It can be seen from the above that, at a control break, totals are printed from the lowest level up to the current break, but that the headings are printed from the highest level downwards. This is an important concept for the printing of *HEAD and *OUT lines.

(7) At the end of the card index, total the amounts for the customer and the current salesperson , then produce a grand total. In FILETAB, this grand total line is known as an *OUT line at the final level (a *OUT F).

This example has assumed throughout that the cards in the manager's index were in the correct order for the report. Had they been in any other order, they would need to be sorted in such a way that the minor control fields were grouped within the major control fields. FILETAB has its own sorting mechanism (by way of the *SORT directive), which allows the hierarchy of control levels to be specified using field specifying characters used in the *INLIST. This use of FSCs is discussed in detail in *Chapter 2.11*.

The following flowchart summarises the sequence of operations of the Fixed Logic. It is not intended as a definitive description.



2.2.1 Notes

(1) *CONTROL BREAK - the user defines in the `*INLIST` which fields are to be used as control fields (see Chapter 2.6).*

(2) *ROLL AND ZEROISE TOTALS - totals are rolled to the next highest level and the totals just output are set to zero.*

(3) *The processing denoted by the last three boxes (those dealing with the test for headings, the printing of headings and the printing of data) is intended only as a rough guide.*

A more detailed description of the Fixed Logic is given in Chapter 4.

2.3 *PROGRAM Directive

The `*PROGRAM` directive enables the program being compiled to be named. `*PROGRAM` has the following format:

```
*PROGRAM name
```

- The first character must be alphabetic
- The remainder must be alphabetic, numeric or a hyphen (-)

When this directive is used, it must appear as the first directive in the source statements.

2.4 *FILE Definition

The `*FILE` directive defines the file from which the report is to be produced. Records are read automatically from this main file into an area of store known as the 'Record Area'.

```
*FILE  localname  keyword1=value1,keyword2=value2,.....etc
NAME=physicalname
```

'`localname`' can be of unlimited length and may contain hyphens. It must however, begin with an alphabetic character. It is used to link the logical file in the program with the physical file (as defined in either the `NAME` keyword or an environment variable of the same name).

A keyword may not appear more than once. The list of keyword/values may be split over more than one input line, if the last value on each split line is terminated by a comma. However, a keyword/value pair must not be split. A keyword may be preceded by one or more spaces, but the keyword/value pair must not contain spaces. Character strings given as values are normally abbreviated to their first character.

Some possible values and defaults for *keywordn=valuen* follow. Only those keyword/value pairs required for simple reports are specified; further keywords may be found in *Chapter 6.*

File Name	
keyword: acceptable value: default:	NAME a file name none (see note below)

Table 2-2: Possible values and defaults for *keywordn=valuen*

The NAME keyword associates the localname specified with the physical filename in value.

Note: There is an alternative method which allows the physical filename to be specified in an environment variable.

Filenames explicitly specified using NAME have precedence over any in environment variables.

Note: If neither method is used a runtime error will be produced.

Record Length	
keyword: acceptable value: default:	RL a positive integer 80

Table 2-3: Number of bytes to be transferred into Record Area

This defines the number of bytes to be transferred into the user's Record Area. If the value given is less than the actual length of the record, the record is truncated. If it is greater, the contents of the Record Area following the record is indeterminate.

Organisation	
Acceptable Values	LINE (OR=Line) INDEXED-SEQUENTIAL (OR=I) DIRECT (OR=D)
default	LINE

Table 2-4: Type of file to be processed

The OR parameter defines the type of file to be processed. For a simple report, this is probably LINE. *See Chapter 6.4* before accessing other file types.

2.4.1 Examples

```
*FILE  LNAME   RL=100,OR=I  NAME=lname.isfile
*FILE  CUSTFL  OR=Line      NAME=custfile.txt
*FILE  STOCKFL OR=L,RL=200  NAME=stkfile.txt
```

2.5 *DICTIONARY Definition

The `*DICTIONARY` allows specific data fields to be named. Each field is defined in terms of its start position, type and length, and this 'Field Definition' indicates which of three areas - Record Area, Work Area, Print Area - contains the field. Here is an example of a `*DICTIONARY` and the terminology used to define it:

```
*DICTIONARY
CURRENT-ACCOUNT = 12/6
NAME            = 20/40
PAY             = -8:8
```

The `*DICTIONARY` directive introduces the Field Definitions.

`CURRENT-ACCOUNT`, `NAME` and `PAY` are user-defined fieldnames.

`12/6`, `20/40` and `-8:8` are Field Definitions.

2.5.1 Record and Work Areas

When FILETAB reads a record from the main file, it places the record in an area known as the Record Area. Data items in this area are referenced using fieldnames or startpositions. The first byte of the Record Area is defined to be byte zero, and the start-positions of fields are defined by their byte displacement from byte zero.

Record Area fields may come from files; these fields can be processed by decision tables, and additional fields may be generated in the Record or Work Areas. Record and Work Area fields are accessed by the Fixed Logic of FILETAB for totalling and printing etc.

The contents of the Work Area is maintained exclusively through actions performed in decision tables.

The Record or Work Area may be used for:

- Processing data
- Temporary storage
- Creating records to be written to output files
- Holding records read from input files

Fields in the Work Area are defined relative to the last byte, which is deemed to be -1. Thus a three byte field in the Work Area could occupy bytes -3, -2, -1.

The addressing convention of the Record and Work Areas can be illustrated as follows:

```
RECORD AREA
0 1 2 3 4 5 etc.
```

```
WORK AREA
etc. -6 -5 -4 -3 -2 -1
```

2.5.2 Format of *DICTIONARY Detail Statements

Following the `*DICTIONARY` directive, input detail statements, in the following format, for each field defined:

```
Fieldname = Field Definition
```

You can specify more than one entry on a line as long as the entries are separated by commas and are not split over succeeding lines. Spaces may be inserted between components. The `*DICTIONARY` detail statements are terminated by the next directive, unless this is `*PAGE` or `*COPY`. The following restrictions apply to fieldname, it must:

- Commence with an alphabetic character
- Contain only alphanumeric characters or hyphens
- End with an alphanumeric character
- Not contain successive hyphens
- Not be a Reserved Word (see *Chapter 7*)
- Not commence with CC1 - 80 or PP1 - PP160

'=' is a separator whose use is optional; if it is omitted, there must be at least one space between the fieldname and the following Field Definition.

Field Definition describes the position, type and length of a field:

```
[start-position]       type       length
```

Here, '`start-position`' may be in the Record, Work or Print Area, and, if specified, must be one of the following:

- An unsigned integer defining a byte position in the Record Area
- A negatively signed integer defining a byte position in the Work Area
- A print position preceded by the letters PP
- A card column number preceded by the letters CC, where CC1 represents byte zero of the Record Area

*Note: IF start_position is omitted the start_position is taken from the end position of the previous DICTIONARY entry (see *OPT ALI description in Chapter 5)*

'`type`' can be:

- `/` for character fields
- `:` for signed binary fields
- `;` for unsigned binary fields

Note: When CC or PP is used, type must be character, with the exception of PP-1 and PP0 which must be a one byte binary.

'`length`' defines the length of the field in bytes.

Note: Field Definitions must not contain embedded spaces.

2.5.3 Examples of *DICTIONARY Detail Statements

Entry	+	Notes
NAMEADD	= 20/80	a character field in the Record Area
VALUE	= 12;4	an unsigned binary field in the Record Area
WK-2	= -8:4	a signed binary field in the Work Area
WK-1	= -3/3	a character field in the Work Area
DEPT	= CC1/4	a character field in the first four bytes of the Record Ar
PRT-FLD	= PP2/4	a character field in the print-line

Table 2-5: Examples of Dictionary detail statements

The `*DICTIONARY` to define the transaction record described in *Example Files on page 6 of Chapter 1* could be:

```
*DICTIONARY
TT          =   0/2
ACCOUNT-NO =   4/4
INVOICE-NO =   8/6
REF-NO      =   14/6
DATE        =   20/6
VALUE       =   28:4
VAT         =   32:4
DISC        =   36:4
ORDNO       =   40/12
PROD-CODE   =   52/3
QTY-CODE    =   55/3
QTY         =   60:4
```

This basic format of Field Definitions allows definition of any type of field in the Record and Work Areas. There are variations which make it easier to define fields in the `*DICTIONARY` described in *Alternative Field Definition Formats on page 1 of Chapter 8.*

2.5.4 Notes on *DICTIONARY

(1) `*DICTIONARY` is an optional directive, its use is strongly recommended in all programs as an aid to documentation and program maintenance.

(2) Fields may be specified in any sequence in the `*DICTIONARY`. To ease maintenance, standards should be adopted. For example, entries might be listed in the sequence of their start-positions, or in the alphabetic sequence of their `*DICTIONARY` names.

(3) Not all fields on a record need to be defined, and padding is unnecessary.

(4) Binary fields normally have a length of four or eight bytes, since these constitute whole words. Binary fields with other lengths perform less efficiently.

Details of the restrictions on field start-positions and lengths may be found in *Restrictions on Field Start-positions and Lengths on page 4 of Chapter 8*.

2.6 *INLIST Definition

The *INLIST directive and associated detail statements define the fields on an input file which are to be printed. They also define the fields for which totals are to be accumulated, and which fields are to control the accumulation of these totals. Thus the *INLIST acts as a link between the input and the report to be produced. The *INLIST directive has no operands.

Here is an example of an *INLIST and the terminology used to define it:

```
*INLIST
A ACCOUNT
B NAME
C 30/20
1 QTY
2 -8:8
```

A,B,C,1 and 2 are field specifying characters (FSCs).

ACCOUNT, NAME and QTY are fieldnames previously defined in a *DICTIONARY.

30/20 and -8:8 are Field Definitions.

Before the format of the *INLIST detail statements is described, two concepts must be understood:

- The definition of a field on the input file as a control, totalling or transfer field
- Field specifying characters

2.6.1 Control, Totalling and Transfer Fields

A 'totalling' field is a numeric field for which totals are to be accumulated. It may be held as characters or binary. The contents of a field defined as a totalling field are automatically accumulated by FILETAB.

A 'control' field is a field which controls the accumulation of totals, and/or controls the layout and format of the report.

A 'transfer' field is a field to be printed on the report, but used neither for control nor totalling purposes.

2.6.1.1 Example

A report is to be produced from the transaction file (see the file layout in *Chapter 1.4.1*) listing Account Number, Product Code, Value, Quantity, Quantity Code and Invoice Number. Totals of Value are to be produced for each Account Number and Product Code. The file is held in Product Code within Account Number sequence.

The Value field would be defined as a totalling field, since totals of this field are required.

Account Number and Product Code would be defined as control fields, since they control the way in which the totals are accumulated. Product

Code would be the minor control field, and Account Number the major control field.

The remaining fields, Quantity, Quantity Code and Invoice Number, would be defined as transfer fields.

Note: While control, totalling and transfer fields often relate to the Record Area, fields may be defined in the Work Area. In the example above, if it were required to produce totals of Quantity, then before totals were accumulated it would be necessary to scale the Quantity field depending on the Quantity Code. This calculation could be performed in the Work Area using a decision table (see Chapter 3). The Quantity field would be defined as a totalling field.

2.6.2 Field Specifying Characters (FSCs)

A field specifying character, usually abbreviated to FSC, is a single character, from the character set available on the computer, which specifies whether a field is to be a control, totalling or transfer field.

The available character set is divided into three groups:

control fields	`M N O P Q R`
totalling fields	`0 1 2 3 4 5 6 7 8 9` `S T U V W X Y Z`
transfer fields	the remainder of the character set that is, `A B C D E F` etc. except for space `' [ * ,`

Table 2-6: Available character set

This allocation of FSCs is the default setting, and should be adequate for most reports. For information on the re-allocation of FSCs, see *Chapter 4*.

A sequence is implicit in the characters used to define control fields. `M` defines the lowest control level, `N` the next highest and so on. It is important that the control level FSCs are defined consecutively, starting at `M`, and intermediate control levels are not omitted. Any of the totalling or transfer FSCs may be used to define totals or transfer fields.

Up to six levels of control may be defined, corresponding to the FSCs `M`, `N`, `O`, `P`, `Q` and `R`. In fact, there are two additional levels: the lowest or 'List' level (`L`), which breaks for every record selected, and the highest or 'Final' level (`F`), which occurs at the end of processing.

The control levels `M` to `R` are user-defined and are linked to fields in the Record and/or Work Areas. The control levels `L` and `F` are system-defined, and their use does not affect the use of `L` and `F` as transfer FSCs. As detailed later, these special List and Final control levels can only be used as operands in `*OUT`, `*HEAD`, `*DETAB BREAK` and `*LOOKUP BREAK` directives to indicate at which control level a line is printed or at which level a decision table is entered.

2.6.3 Format of *INLIST Detail Statements

The detail statements which follow the *INLIST directive define each field in one of the following formats:

```
1   FSC   fieldname
2   FSC   Field Definition
```

'*FSC*' is a control, totalling or transfer field specifying character.

'*fieldname*' is previously defined in a *DICTIONARY or is a Reserved Word.

'Field Definition' is as described in Chapter 2.5 with certain restrictions, described in Chapter 4.

Note: A further, third, format, in which no fieldname or Field Definition follows the FSC, is also available. It is known as a 'dummy totalling field', and this too is described in Chapter 4.

One of the above entries is made for each field defined in the *INLIST. More than one entry may be specified on a line; *details may be found in Chapter 4*. Spaces may be inserted between components. Entries may be specified in any sequence. The *INLIST detail statements are terminated by the next directive, unless this is *PAGE or *COPY.

2.6.3.1 Example

The *INLIST for the transaction record defined earlier in this Chapter could be written in format type 1 as:

```
*INLIST
S VALUE         [ totalling field
M PROD-CODE     [ control field
N ACCOUNT-NO    [ control field
A QTY           [ transfer field
B QTY-CODE      [ transfer field
C INVOICE-NO    [ transfer field
```

Whereas format type 1 above links an FSC to the name of a field in the Record or Work Area, format type 2 links an FSC directly to a field in the Record or Work Area.

2.6.3.2 Example

The *INLIST above can be written using format type 2 as:

```
*INLIST
S 28:4
M 52/3
N 4/4
A 60:4
B 55/3
C 8/6
```

For 'one-off' programs, this format may provide the programmer with a useful shorthand, but once again the use of a *DICTIONARY is strongly recommended.

2.6.4 *INLIST Area

Fields defined in the *INLIST are moved automatically, by the Fixed Logic, from the Record and Work Areas to an area known as the '*INLIST Area', where accumulation of totals is carried out. All totalling fields are stored in the *INLIST Area as signed binary fields, regardless of their original type. They are stored as either double or single word fields (:4 or :8) depending upon the default FSCT settings (see Chapter 5.2.3) and any individual characteristic alterations that have been defined (see Chapter 4.1.1).

2.7 Input Specification Summary

So far, the use of four directives has been covered:

(1) The *PROGRAM directive names the program.

(2) The *FILE directive and its associated operands define the file from which the report is to be produced. FILETAB automatically reads records from this file into the Record Area.

(3) The *DICTIONARY directive and its associated detail statements link user-defined names with fields in the Record and Work Areas.

(4) The *INLIST directive and its detail statements define whether fields in the Record or Work Areas are used for control, totalling or transfer purposes.

2.7.1 Example

It is required to produce a report from the transaction file listing, for each record; the Area Code, Account Number, Invoice Number, Date, Value and Quantity. Quantity is to be printed as the product of Quantity and Quantity Code.

Totals of Value and Quantity are to be printed at change of Account Number and Area. The file is in Account Number within Area sequence.

The source statements are as follows:

```
*PROGRAM EXAMPLE3
*FILE TRANSDY  NAME=transdy.txt
*DICTIONARY
TT = 0/2,         AREA = 2/2
ACCOUNT-NO = 4/4, INVOICE-NO = 8/6
INV-DATE = 20/6,  VALUE = 28:4
QTY-CODE = 55/3,  QTY = 60:4
QTYACT = -8:8
*INLIST
M ACCOUNT-NO
N AREA
A INVOICE-NO
B INV-DATE
1 VALUE
2 QTYACT
*DETAB RECORD
```

```
A QTYACT MV QTY                [ MOVE QUANTITY TO WORK FIELD
  QTYACT *  QTY-CODE           [ MULTIPLY WORK FIELD BY CODE
*GO
```

2.7.2 Notes

(1) Account Number is the lower control level, and Area the higher level (remember that `M`, `N`, `O`, `P`, `Q` and `R` are in ascending sequence).

(2) Invoice Number and Date are transfer fields.

(3) The quantity to be printed has to be calculated by multiplying Quantity by Quantity Code. This is done in the decision table `*DETAB RECORD`, and a Work Area field (`QTYACT`) has been reserved for the result of this calculation. Printing and totalling is carried out using this field, which is defined by the totalling FSC 2.

(4) In the `*DICTIONARY`, some lines contain two entries. This is quite permissible, provided that the entries are separated by commas.

The use of decision tables is described in *Chapter 3.*

2.8 Printing Headings

2.8.1 General

FILETAB allows considerable flexibility in the format and position of report headings. Headings may be printed at the top of every page, whenever a certain number of lines have been printed on the previous page, or when a control field changes. Current date, time and automatic page numbering may be included in headings. Data from the file being printed may also be included in the heading. A report title can be printed, appearing once only, on the first page of a report. Here is a section of a FILETAB report illustrating these points:

```
03/01/01  LIST OF TRANSACTIONS FOR AREA 01  PAGE    1

AREA   ACCOUNT      INVOICE NO. DATE          VALUE
-----------------------------------------------------
01     5016         AB0445      21/01/94    1120.00
01     5016         AB0445      21/01/94      31.50
01     5016         AB0910      25/01/94     234.50
01     5016         DF2103      14/04/94      15.10
01     5016         DF2214      15/04/94     421.12

       TOTAL FOR ACCOUNT 5016               1822.22

01     8002         AB1246      21/01/94     130.75
01     8002         AD1278      25/01/94      49.60
01     8002         AC1402      11/02/94      50.10
01     8002         AF2100      14/03/94     250.00

       TOTAL FOR ACCOUNT 8002                480.45

       AREA 01 TOTAL                        2302.67
```

In the above example, Account and Area are control fields (Area being the higher or major control field). These control fields control the totalling of the field Value, and the layout of the report. Other fields on the report are transfer fields. The lines above, including the line of dashes, constitute the heading of the report, and are printed when a page is full, or when the major control field, Area, changes. Headings are specified in FILETAB using the `*HEAD` directive, which is followed by detail statements describing the contents of the heading required, in a picture-image format:

```
*HEAD L CH1,2
ANALYSIS OF SALES BY AREA

AREA    PRODUCT CODE   QUANTITY    VALUE
```

Operands on the `*HEAD` directive specify when and where on a page the heading is to be printed. A title may be added to the report using the `*TITLE` directive, which is followed by detail statements describing the contents of the title required, in a picture-image format:

```
*TITLE
THIS REPORT IS FOR THE ATTENTION OF M.V. SMUTHERS
```

2.8.2 Format of *HEAD and *TITLE Directives

The general formats of the `*HEAD` and `*TITLE` directives are:

```
*HEAD    operands
detail statement(s)
*TITLE   operands
detail statement(s)
```

The detail statements are images of the headings required; thus in the above `*HEAD` example, when the line:

```
ANALYSIS OF SALES BY AREA
```

is input, if the word `ANALYSIS` is preceded by two spaces. When the heading, is output on the line-printer, it is preceded by two spaces. If multi-line headings are required, as in the above example, each line consists of pairs of input details with the first detail representing print positions 1 to 80, and the second detail representing positions 81 to 160. When an odd number of input details is used, FILETAB pads with spaces.

Each `*HEAD` or `*TITLE` directive may be followed by up to 120 detail statements, defining 60 lines of print.

Note: See *Chapter 2.1.5* for further details

The detail statements are terminated by the next directive, unless this is `*PAGE` or `*COPY`.

2.8.3 Including Date, Time, Page No. and Mark No. in Headings

FILETAB recognises certain character strings, as a request to print current date, time, page number and mark number when included in `*HEAD` detail statements. These strings are:

```
DD/MM/YY   Current date
HH.MM.SS   Current time
PPPP       Page number
MKMK       FILETAB program mark number
```

2.8.3.1 Example

```
*HEAD L CH1,2
SALES ANALYSIS ON DD/MM/YY AT HH.MM.SS        PAGE PPPP
```

appears on the report as:

```
SALES ANALYSIS ON 03/01/01 AT 16.20.35        PAGE    7
```

2.8.4 Including Data in Headings

Data on the file which is being printed may be included in headings. The field to be included in the heading is given an FSC in the usual way using the `*INLIST` directive. A string of these FSCs is then specified in the appropriate position in the heading, and enclosed within single quotes. For example, a listing from CUSTFILE (see Chapter) includes the Credit Code and Account Number in the page heading. The relevant source statements are:

```
*DICTIONARY
CRCODE = 328/2
ACCNO  = 4/4
*INLIST
N CRCODE
M ACCNO
*HEAD L CH1,2
LISTING OF ACCOUNT'MMMM' CREDIT CODE 'NN'
```

and the heading produced:

```
LISTING OF ACCOUNT 4320  CREDIT CODE  50
```

2.8.4.1 Notes

(1) The strings of FSCs must be enclosed in single quotes. If they had been omitted in the above example, FILETAB would have printed strings of `M`s and `N`s in the relevant positions.

(2) The quotes are replaced by spaces on the report.

(3) The use of single quotes as apostrophes is not allowed. The following example, for instance, is invalid:

```
*HEAD L CH1,2
LISTING OF MAINTENANCE DIVISION'S STAFF
```

(4) The number of FSCs listed for character fields should equal the field length defined in the `*DICTIONARY`. In the example, the maximum size of credit code is 99, so two characters have been reserved for the field by the string 'NN'. When insufficient FSCs are specified for character control and transfer fields, characters are lost from the right-hand end. For other types of field, truncation is at the left-hand end.

Note: For numeric fields, only an insignificant or zero portion of the field may be truncated.

(5) Only transfer and control FSCs may be used when inserting data into headings. A transfer field printed in a heading at a break level higher than `L` contains the appropriate value from the first record of the control group.

2.8.5 Operands on *HEAD Directive

When the format of the `*HEAD` directive was described, it was noted that operands could be added to specify when and where on a page a heading is to be printed.

The operands on the `*HEAD` directive are:

```
*HEAD  control-levels  b,a,p
```

`control-levels` represents the control level list, and consists of one of the control level characters `M` to `R`, or `F` (Final level) and/or the special List control level character `L`, which means that the heading is printed on every page. The control level list indicates at which control level(s) a heading is to be printed.

For example, if a control level of `M` is specified, and `M` is associated with the Product Code in the `*INLIST` statements:

```
*INLIST
M PRODCODE
*HEAD M 2,1
  PRODUCT CODE ANALYSIS etc.
```

The heading is printed when the Product Code changes, after the printing of totals for the previous Product Code group.

If the special control level `L` is specified, printing of the heading depends on the value of the `b` operand (*see below*).

If one of the control FSCs `M` to `R` is coupled with the `L` control level (for example, `*HEAD L,M`), the heading is printed either for each record / new page taken (`*HEAD L`), or when the control field defined by the FSC (`M` in the example) changes. Where both conditions apply, the heading is printed only once.

Note: Where multiple control-level FSCs are specified in the `*HEAD` and `*OUT` directives, they are separated by commas. No spaces should be included between the comma and the next FSC.

If the special control level `F` is specified, the heading is printed at the beginning of the run.

When it appears on a `*HEAD L,b` may be specified either as simply the number of lines to be spaced before the heading is printed, in which case the heading is printed for every record moved into the `*INLIST` Area, or as `CHn`, where 'n' is the line number at which it is to be printed, counting from line 0. In this case, it is printed only when a new page is taken.

When it appears on a *HEAD at any level other than L, b may be specified in either of the two ways noted above, but in either case, it is only printed when the associated control field changes.

a is the number of lines to be spaced after the heading.

p is the required page size (in lines).

The heading is printed after the spacing or positioning specified by b. After printing, the spacing specified by a is obeyed. In the above example, two lines would be spaced before the heading was printed, and one line after.

If b had been specified as CH0 (that is, *HEAD M CH0,2), a throw to the top of the next page would have been performed before the heading was printed. Two blank lines would have been spaced after the heading.

Further details of b, a and p, and further information on headings, may be found in Chapter 4.

2.9 Printing Data

2.9.1 General

FILETAB allows considerable flexibility in the formatting, layout and editing of data printed on reports.

The report data format is specified using the *OUT directive followed by detail statements which describe it in a picture-image format. The position of a data field on a line is indicated by the use of a string of the FSCs which define that field in the *INLIST. Editing characters may be inserted where required in the string of FSCs. For example:

```
*DICTIONARY
CUSTNO = 2/6
DATE   = 20/6
VALUE  = 28:4
*INLIST
A CUSTNO
B DATE
1 VALUE
*OUT L 1,0
  AAAAAA    BB/BB/BB   £11111.11
```

2.9.2 *OUT Detail Statements

The detail statements are images of the output layout required. Strings of FSCs show where the fields to which they refer in the *INLIST are to be printed. In the above example, the string AAAAAA shows where the Customer Number is to be printed.

Pairs of detail statements represent the print-line image, the first detail representing print positions 1 to 80, and the second detail representing positions 81 to 160. When an odd number of input details is used, FILETAB pads with spaces.

Each *OUT directive may be followed by up to 120 detail statements, defining 60 lines of print.

Note: See *Chapter 2.1.5 for further details.*

A set of detail statements is terminated by the next directive, unless this is *PAGE or *COPY.

The example which follows indicates the use of *OUT detail statements.

2.9.2.1 Example

The *DICTIONARY, *INLIST and *OUT directives to print the Account Number, Name and Address from the CUSTFILE file are:

```
*DICTIONARY
ACCNO    = 4/4
NAME     = 8/36
ADDRESS  = 44/104
*INLIST
A NAME
B ACCNO
C ADDRESS
*OUT L 2,0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA       BBBB

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

CCCCCCCC
```

2.9.3 Control and Transfer Fields

If insufficient FSCs are specified for character control or transfer fields, these fields are truncated from the right when they are printed.

If too many FSCs are specified for control and transfer fields, then FILETAB fills the remaining FSCs with characters from the sending field. The sequence of characters in the sending field is repeated until all remaining FSCs have been filled.

Example

```
*DICTIONARY
NAME = 4/4
*INLIST
A NAME
*OUT L
AAAAAAAAAAA
```

If NAME contains 'ABCD', the *OUT L, when printed, is:

```
ABCDABCDABC
```

When a transfer field is included in an *OUT statement at any break level higher than L, it contains the appropriate value from the last record of the control group.

2.9.4 Totalling Fields - Editing Characters

The characters `/.,` and `space £ $ + - *` may be used to edit totalling FSC strings, and are printed wherever they occur, unless insignificant. Further details may be found in *Chapter 4.*

2.9.5 Totalling Fields - Floating Symbols

Up to three of the characters `£ $ + -` and `*` may immediately precede a totalling FSC string. These characters are floated past leading spaces in the substituted field. Further details may be found in *Chapter 4.*

2.9.6 Incorporation of Literals in *OUT Detail Statements

Literals which are to be included in `*OUT` detail statements must be enclosed within single quotes:

```
*OUT L 1,0
'DATE'  AA/BB/CC 'VALUE' £1111.11
```

Note: This is different from `*HEAD`, where the FSCs are surrounded by quotes.

2.9.7 *OUT Operands

This directive is used to specify the control level(s) and line spacing to be used when printing. The format is:

```
*OUT    control-levels    b,a,p
```

`control-levels` represents the control level list, and consists of one or more of the control level characters `M` to `R` and/or the special control level characters `L` and `F`. *Further details of the control level list may be found later in this section.*

Note: Where multiple control-level FSCs are specified in the `*HEAD` and `*OUT` directives, they are separated by commas. No spaces should be included between the comma and the next FSC.

`b` is the line spacing performed before the output details are printed.

`a` is the line spacing performed after the output details are printed.

`p` is the required page size (in lines).

Further details of `b`, `a` and `p` appear in Chapter 4.

The control level list is used to indicate the control level at which a particular detail format is to be used. For example, if an `*INLIST` is specified as:

```
*INLIST
M ACCNO
N AREA
```

then the details following an `*OUT M` are printed when Account Number changes, and any totals specified in the details are the accumulated totals

for that Account Number. The details following an `*OUT N` are likewise printed when Area changes.

The control levels `L` and `F` (List and Final), when specified as part of the control level list operand, indicate that the following detail format is to be used for each record printed (List Level), or for the final overall totals (Final Level), when the end of the run is reached.

2.9.7.1 Examples

```
*OUT    L    0,1
```

This directive prints its associated details for each record, no lines being spaced before printing and one after.

```
*OUT    M    2,1
```

This directive prints its associated details every time the field defined by the control FSC `M` changes, two lines being spaced before printing and one after.

2.10 Output Specification Summary

Three directives - `*TITLE`, `*HEAD` and `*OUT` - enable description of the report layouts. The formats of these directives are:

```
*TITLE  b,a
detail statements
*HEAD  control-levels  b,a,p
detail statements
*OUT  control-levels   b,a,p
detail statements
```

`detail statements` describe the contents of lines to be printed, using a picture-image format. For `*OUT`, the detail statements usually consist of strings of FSCs, while for `*HEAD` and `*TITLE` the detail statements usually consist of the appropriate text. Data may be included in headings by enclosing strings of FSCs in single quotes. Literals included in detail output must be enclosed by single quotes. Multiple details, each pair of details corresponding to one line of output, may be specified for `*HEAD`, `*TITLE` and `*OUT`. A series of detail statements is terminated by the next directive, unless this is `*PAGE` or `*COPY`.

`control-levels` indicate the control level at which a particular heading or detail output is printed. For `*HEAD`, control-levels may be L and/or one of the control FSCs `M`, `N`, `O`, `P`, `Q` and `R`. For `*OUT`, control-levels may be one or more characters from `L`, `M`, `N`, `O`, `P`, `Q`, `R` and `F`.

Note: Where multiple control-level FSCs are specified in the `*HEAD` and `*OUT` directives, they are separated by commas. No spaces should be included between the comma and the next FSC.

`b` and `a` indicate line spacing before and after printing.

`p` indicates the required page size (in lines), and is taken from the last `*HEAD` or `*OUT` directive to specify it.

Further details of `b`, `a` *and* `p` *may be found in Chapter 4*

The following example defines a report from the CUSTFILE file (*see Chapter 1.4.1*), illustrating the above points.

```
*PROGRAM EXAMPLE4
*FILE CUSTFILE RL=332  NAME=custfile.txt
*DICTIONARY
CUSTNO      = 4/4
AREA        = 2/2
NAME        = 8/36
OUTSBAL     = 324:4
CRCODE      = 328/2
*INLIST
A NAME
B CUSTNO
M CRCODE
N AREA
1 OUTSBAL
*HEAD  L,N CH1,1,55
DD/MM/YY  OUTSTANDING BALANCES FOR AREA 'NN'       PAGE PPPP

CREDIT  CUSTOMER NAME                             OUTSTANDING

CODE    NO                                            BALANCE

*OUT L  1,0
    MM  BBBB AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA   £1111.11
*OUT M  2,0
       'TOTAL FOR CREDIT CODE'MM                   £11111.11
*OUT N  1,0
       'AREA TOTAL'                               £111111.11
*GO


03/01/01  OUTSTANDING BALANCES FOR AREA  01        PAGE    1
CREDIT  CUSTOMER NAME                             OUTSTANDING
CODE    NO                                            BALANCE

    10  0101 FRED BROWN AND COMPANY                   £149.56
    10  6540 STIRLING INVESTMENTS                     £645.00

        TOTAL FOR CREDIT CODE 10                      £794.56

    20  9671 XAVIER RUBBER COMPANY LTD.              £1250.00
    20  9876 ZEBRA HOLDINGS                            £45.21

        TOTAL FOR CREDIT CODE 20                     £1295.21
        AREA TOTAL                                   £2089.77
```

Note: The file CUSTFILE must contain records in the sequence of Credit Code within Area, or the tabulation process generates totals indiscriminately.

2.11 *SORT Directive

The `*SORT` directive is used when the output sequence of data is to differ from the input sequence.

When specified, the *SORT directive must appear after the *INLIST directive (though not necessarily immediately after). Only one *SORT can appear in a program. The format of the *SORT directive is:

```
*SORT
```

OR

```
*SORT sortkey1,sortkey2 etc
```

If the *SORT directive is specified without sortkeys, sorting takes place according to the control fields specified in the *INLIST. For example, if the control levels M, N and O were specified in the *INLIST, the assumed *SORT would be:

```
*SORT *O,*N,*M
```

*Note: If there are no control fields, and no sortkeys are specified on the *SORT directive, no sorting takes place.*

If sortkeys are specified, the most significant key must be specified first. The sortkeys may be any of the following:

```
fieldname from *DICTIONARY
start-position type length
*FSC
```

Unless otherwise specified the sort is in ascending order. Suffixing a sortkey with a minus (-) reverses the sort order for that sortkey. Suffixing the *SORT command with a minus reverses the sort order for all of the following sortkeys (that is, descending). The two forms can be combined. In the following example sortkeys 1 and 3 are in descending order and sortkey 2 in ascending order:

```
*SORT- sortkey1,sortkey2-,sortkey3
```

Note: Numeric fields are sorted numerically, therefore in ascending order, negative numbers appear before positive numbers. In some other environments, such as VME, numerics are treated as characters and therefore negatives appear after positive numbers.

2.11.1 Examples

```
*SORT
*SORT ACNO
*SORT AREA,ACNO
*SORT *A,-24/3
```

2.11.2 Operation of the Sort

For each record selected, a sort record is formed, containing only the sortkey(s) and the fields defined in the *INLIST (which means that the sort is performed only on the information required for output). Dummy totalling fields (*see Chapter 4*) are not included in sort records. These sort records are built up in store until the available space is full, and then written to the sort workfile; if the end of the Main File is reached before the store space is full, the entire sort takes place in store.

The sort itself is entered only when all processing at RECORD and ENDFILE Decision Points has been completed. *See 4.2 for a description of these Decision Points.*

2.12 *END, *GO and *STOP Directives

A set of FILETAB source statements is terminated by `*END`, `*GO` or `*STOP`. Any of these directives indicates that the source program is complete.

Further details of these directives appear in Chapter 5.

2.13 Worked Example

This example illustrates the features of FILETAB described in this chapter.

A report is produced showing Area, Account Number and Balance for each customer whose details are held on the customer master file. Balance totals appear for each Area and at the end. Records are to be printed in the sequence of Account Number within Area.

```
*PROGRAM EXAMPLE5
*FILE CUSTFILE RL=332  NAME=custfile.txt
*DICTIONARY
AREA    = 2/2   [ AREA
ACCNO   = 4/4   [ ACCOUNT NUMBER
OUTSBAL = 324:4 [ BALANCE
*INLIST
M AREA
A ACCNO
2 OUTSBAL
*HEAD L CH1,2
DD/MM/YY     CUSTOMER BALANCE LISTING     PAGE PPPP



AREA     ACCOUNT             BALANCE

         NUMBER

*OUT L 1,0
  MM       AAAA            £22222.22
*OUT M 2,1
'AREA'MM'TOTAL'           £222222.22
*OUT F 3,0
'FINAL TOTAL'            £2222222.22
*SORT  AREA,ACCNO
*GO
```

A sample of the output appears below:

```
03/01/01     CUSTOMER BALANCE LISTING     PAGE    1

AREA     ACCOUNT             BALANCE
         NUMBER
```

```
 21       1144             £1511.20
 21       1313              £151.50
 21       8643              £109.22
 21       9187               £12.50
 21       9221               £90.50
 21       9768              £200.30
 21       9827              £210.80

AREA 21 TOTAL              £2286.02

 22       3211               £16.84
 22       3270              £128.62

FINAL TOTAL               £74210.23
```

Chapter 3
Decision Tables

All program logic, processing, record selection etc. in FILETAB must be specified in decision tables. The first section of this chapter introduces decision table theory, and the remainder describes the use of decision tables in FILETAB.

3.1 Decision Table Theory

A decision table is a method of showing, in a tabular form, the actions to be carried out for different combinations of conditions.

3.1.1 Examples

3.1.1.1 Example 1

For example, the following decision table illustrates which London underground routes one could follow to reach various attractions, assuming that the journey commences at Leicester Square.

Destination?						
Buckingham Palace	Yes					No
Houses of Parliament		Yes				No
Regent’s Park			Yes			No
St. Paul’s Cathedral				Yes		No
10 Downing Street					Yes	No
Route						
Piccadilly to Piccadilly Circus			X			
Piccadilly to Holborn				X		
Northern to Embankment	X	X			X	
Change Tubes	X	X	X	X	X	
Bakerloo to Regents Park			X			
Central to St. Paul’s				X		
District/Circle to Westminster		X			X	
District/Circle to St. James’s Park	X					
Ask a policeman						X

Table 3-1: Routes from Leicester Square

The table is given a name, 'Routes from Leicester Square'. This is to distinguish it from other decision tables, for example, 'Routes from Oxford Circus'.

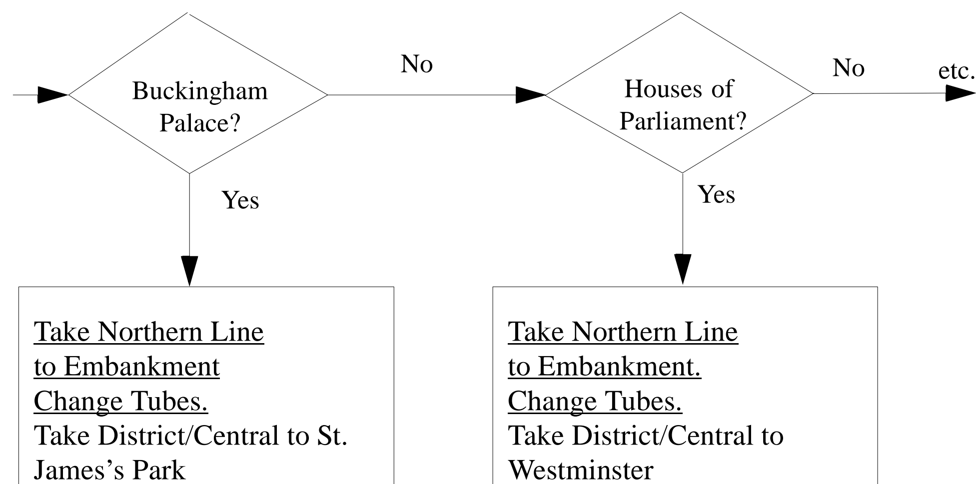
The decision table is divided into four sections:

Condition stub Consists of a list of the conditions to be evaluated	**Condition entries** Consists of a list of the responses to the conditions, usually Yes and No (or `Y` and `N`)
Action stub Consists of a list of the actions to be carried out	**Action entries** Consists of a list of the particular actions to be carried out, they are indicated by Xs placed alongside the appropriate actions

Thus *Table 3-1* indicates that to get to St. Paul's Cathedral from Leicester Square, the Piccadilly line should be taken to Holborn station, followed by a change to the Central line, taken as far as St. Paul's station.

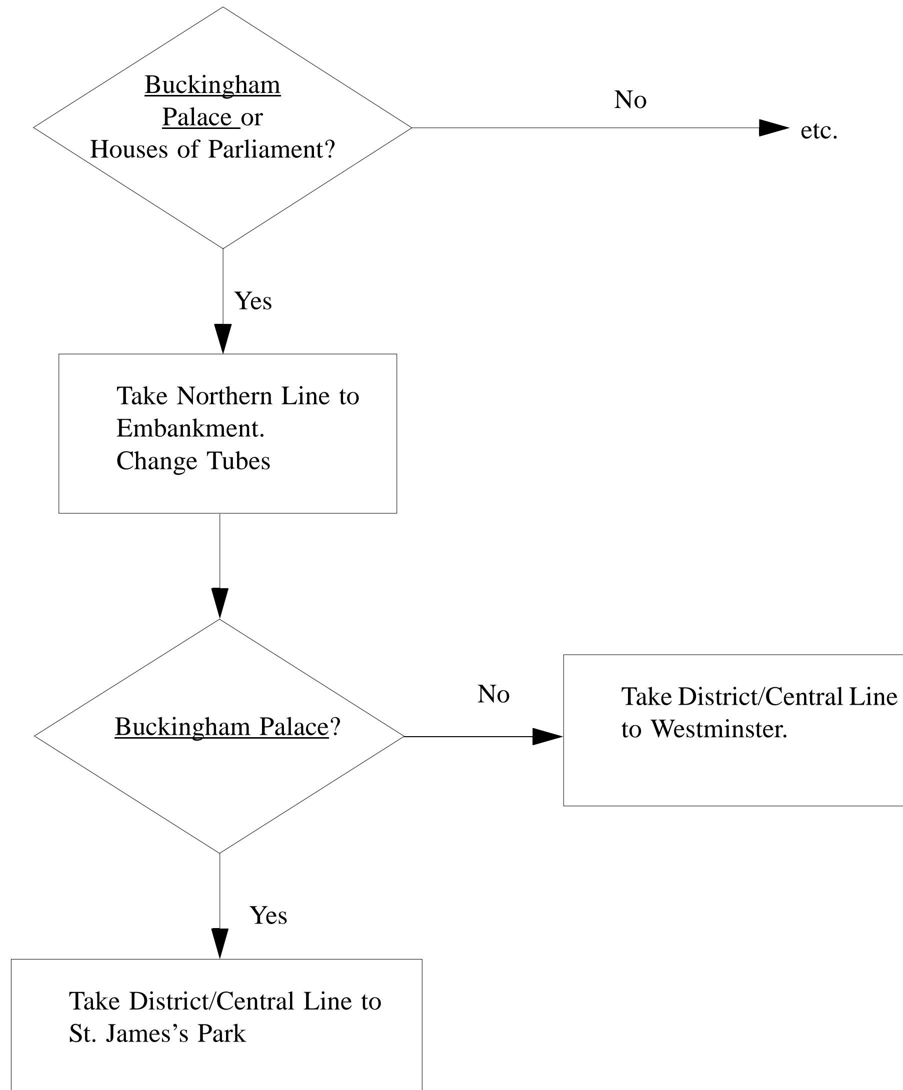
The entry part of the table is further subdivided into a number of vertical columns, each is known as a rule. A rule corresponds to a particular combination of conditions.

The conditions (destinations in this example) are listed separately from the actions (the routes). This has the advantage of making decision tables a more concise method of expressing logic than narrative description or a flowchart, since repetition of conditions and actions is avoided. For example, consider a flowchart which shows how to get to Buckingham Palace or the Houses of Parliament (the case covered by rules 1 and 2 of the decision table).



The underlined actions are repeated.

To avoid repetition of these actions, they can be combined as follows:



Here, repetition of actions has been avoided, but the underlined condition is repeated. Similar repetition occurs when the problem is expressed in narrative form.

3.1.1.2 Example 2

Decision table 'Order Processing'				
	1	2	3	4
Quantity ordered < or = Order limit	Y	Y	Y	N
Credit approval	Y	Y	N	.
Quantity in stock > or = Quantity ordered	Y	N	.	.
Dispatch quantity ordered to customer	X	.	.	.
Dispatch available stock to customer	.	X	.	.
Notify accounts dept. of dispatch	X	X	.	.
Re-order stock	.	X	.	.
Reject order	.	.	X	X

Table 3-2: Order Processing

The Yes and No entries are specified in the condition entry as `Y` and `N`. A dot (or dash) in the condition means that the result of evaluating a condition is immaterial for that rule. For example, in rule 3 it is immaterial whether quantity in stock is greater than or equal to quantity ordered, since the order is going to be rejected. A dot (or dash) in the action entry means that the action is not to be performed for that rule.

The relationship between the conditions in a rule is an `AND` relationship, that is,

```
IF     Condition  1     does/does not apply
AND    Condition  2     does/does not apply
AND    Condition  3     does/does not apply
THEN   Action 1         is/is not executed
AND    Action 2         is/is not executed
etc.
```

In *Table 3-2*, the relationship between the rules (vertical columns) is an OR relationship, that is,

```
    Rule 1    is satisfied
OR  Rule 2    is satisfied
OR  Rule 3    is satisfied
OR  Rule 4    is satisfied
```

Note: Decision tables may be constructed with an `AND/OR` *relationship between rules, that is,*

```
        Rule 1    is satisfied
AND/OR  Rule 2    is satisfied
etc.
```

In this type of decision table, more than one rule may be satisfied.

3.1.2 ELSE Rule

In decision tables, the same actions are often to be carried out for different rules. When this is so, rules which have the same actions may be grouped together. One way to do this is to use an `ELSE` rule.

Example 2 on page 4 can be re-written by combining rules 3 and 4 into an ELSE rule.

3.1.2.1 Example 3

Quantity ordered < or = Order limit	Y	Y	ELSE
Credit approval	Y	Y	.
Quantity in stock > or = Quantity ordered	Y	N	.
Dispatch quantity ordered to customer	X	.	.
Dispatch available stock to customer	.	X	.
Notify accounts dept. of dispatch	X	X	.
Re-order stock	.	X	.
Reject order	.	.	X

Table 3-3: Order Processing using ELSE

The ELSE rule applies when no other rule in the table is satisfied. It is always written as the last rule in a decision table, and no condition entries except dot (or dash) are specified for the remaining conditions.

In practice, the ELSE rule is used in the following circumstances:

- When the same actions are carried out for different rules
- To detect errors, either in the logic of the decision table or in the data presented to it (see *Example 4 on page 5*)
- To cater for situations not specifically covered; in *Examples on page 1* for instance, the final rule could have been an ELSE rule

To filter out a group of transactions (see *Example 5 on page 6*).

3.1.2.2 Example 4

Update					
Transaction Key > Master Key	Y	N	N	N	ELSE
Transaction Key = Master Key	N	Y	Y	N	.
Transaction Key < Master Key	N	N	N	Y	.
Transaction Key = Insertion	.	N	N	Y	.
Transaction Key = Deletion	.	Y	N	N	.
Transaction Key = Amendment	.	N	Y	N	.
Write Master record to output	X	.	.	.	.
Write Transaction record to output	.	.	X	X	.
Perform error routine	.	.	.	.	X
Read next Master record	X	X	X	.	.
Read next Transaction record	.	X	X	X	X
Repeat	X	X	X	X	X

This decision table represents a simple update, where a master file is updated by records on a transaction file and an updated master file is produced. 'End-of-file' conditions are ignored, and it is assumed that there are no more than one transaction record per key. The ELSE rule is entered

whenever an invalid combination of conditions is satisfied (for example, a deletion for which there is no matching master record).

3.1.2.3 Example 5

Test 05				
Record type = 05	Y	Y	Y	ELSE
Quantity less than 2500	Y	N	.	.
Value greater than 0	Y	.	N	.
Go to valid	X	.	.	.
Go to error	.	X	X	.
Go to others	.	.	.	X

Here, only records of type 05 are validated. The `ELSE` rule is used to pass all other record types to table ’others’.

In this example, if rules 2 and 3 are both satisfied, then, by convention, the action ’Go to error’ is carried out only once. If an action is to be performed twice, then it must be specified twice.

3.1.3 Types of Decision Table

The tables shown in the examples so far are known as ’limited entry’ tables because the values in the entry part of the table are limited to:

```
Y N . (or -) X ELSE
```

Decision tables can be constructed where only part of a condition or action is specified in the stub part of the table, the remainder of the condition or action being extended into the entry part of the table. This type of decision table is known as ’extended entry’. The condition part of Example 4 can be written in extended entry format.

3.1.3.1 Example 6

Trans. key ? Master key	>	=	=	<	EL
Trans. type ?	.	Delete	Amend	Insert	.

Here, `?` is used in the stub to indicate that entries are extended. The action part of the table remains in limited entry. A table such as this, which contains both extended and limited entry lines, is known as a ’mixed entry’ table.

3.1.4 Initial Actions

The decision tables considered so far consist of conditions followed by actions. It is sometimes useful to be able to specify actions which are to be carried out before the conditions are evaluated. These ’initial actions’ are listed in the decision table stub preceding the conditions:

3.1.4.1 Example 7

INITIAL ACTIONS Read next record Space-fill print area					
CONDITIONS End of file Record type = ?	N I	N D	N A	Y .	EL .
ACTIONS Go to ?	Insert	Delete	Amend	End	Er

3.1.5 Linking Decision Tables

Sometimes the sequence of evaluating conditions followed by performing actions is too simple to allow the required logic to be specified. For instance, depending on the values obtained after performing the actions, further conditions may need to be evaluated. To enable complex logic to be specified in relatively simple decision tables, tables may be linked in using `Go to`, `Call` and `If`.

3.1.5.1 Go To

This transfers control to another decision table, which is then obeyed. At the end of this decision table, control is NOT returned to the one which issued the `Go to`:

```
'VALIDATE PAY'
Conditions         Quantity numeric      Y    ELSE
                   Quantity positive     Y      .
                   Quantity < 10000      Y      .
Actions            Go to 'VALIDATE TAX'  X      .
                   Go to 'ERROR'         .      X

'VALIDATE TAX'
Conditions         Tax = 0%              Y      .    ELSE
                   Tax = 8%              .      Y      .
Actions            Go to 'ERROR'         .      .      X
                   Accept data           X      X      .

'ERROR'
Actions            Notify error
                   Reject data
```

3.1.5.2 Call

This transfers control to another decision table, which is then obeyed. At the end of this decision table, control is returned to the action following the `Call`:

```
'CALCULATE PAY'
Condition          Weekly paid                   Y    N
Actions            Call 'COMPUTE OVERTIME'       X    .
                   Add overtime pay to basic     X    .
```

```
'COMPUTE OVERTIME'
Condition         Management grade              Y    N
Action            Overtime hours x 1.5          X    .
                 Pay = overtime hours x rate    X    X
```

3.1.5.3 If

This is a special `Call` made from within the conditions part of a decision table. It is useful for removing complex sets of conditions into another decision table to make the logic simpler. `If` passes control to another decision table, which is then obeyed. At the end of this decision table, an answer ('Exit True' or 'Exit False') is returned to the `If` statement:

```
'AUTHORISE PENSION'
Condition             If 'OAP'            Y   N
Actions               Pay pension         X   .

'OAP'
Conditions            Sex = ?          Male   Female  ELSE
                      Age over ?         65   60      .
Action                EXIT ?           TRUE   TRUE    FALSE
```

3.1.6 Recursion

Decision tables may be entered recursively; a table may re-enter itself until a particular condition is satisfied. One way of doing this is to use `Go to` (*see Go To on page 7*) causing the decision table to be executed again:

```
'LOOP'
Initial action       Read a record
Condition            End of file                 Y      N
Actions              List record on printer      .      X
                     Go to LOOP                  .      X
                    Stop                         X      .
```

Another is to use the action `Repeat`. This returns control to the first condition of the current decision table, missing out any Initial Actions:

```
'LOOP 2'
Initial action      Read first record from the file
Condition           End of file                  Y     N
Actions             List record on printer       .     X
                    Read next record             .     X
                    Repeat                       .     X
                   Stop                          X     .
```

3.1.7 Constructing Decision Tables

Decision tables may be constructed in many ways. The following method is suggested as being particularly suitable for FILETAB decision tables.

(1) List the conditions in any sequence.

(2) List the actions in the correct sequence.

(3) Fill in the rules one rule at a time, until all possible combinations have been catered for.

(4) Combine rules and add `ELSE` rule if appropriate.

(5) Check and simplify the decision table.

Simplifying a decision table consists of merging rules with the same actions into a single rule. Any two rules which have entries identical but for one condition row containing a `Y` and an `N`, may be consolidated into one rule which contains a dot (or dash) in place of the `Y` and the `N`. For example, consider the following table.

```
                        1  2  3  4  5  6  7  8
Credit limit exceeded   Y  Y  Y  Y  N  N  N  N
Prompt payer            Y  Y  N  N  Y  Y  N  N
Special clearance       Y  N  Y  N  Y  N  Y  N
Accept order            X  X  X  .  X  X  X  X
Reject order            .  .  .  X  .  .  .  .
```

Rules 1 and 2 are identical except for a `Y N` pair in row 3, so these two rules may be combined. Similarly, rules 5 and 6, and rules 7 and 8, may be combined as follows:

```
                        1  2  3  4  5
Credit limit exceeded   Y  Y  Y  N  N
Prompt payer            Y  N  N  Y  N
Special clearance       .  Y  N  .  .
Accept order            X  X  .  X  X
Reject order            .  .  X  .  .
```

The table can be further simplified by combining rules 4 and 5 above:

```
                        1  2  3  4
Credit limit exceeded   Y  Y  Y  N
Prompt payer            Y  N  N  .
Special clearance       .  Y  N  .
Accept order            X  X  .  X
Reject order            .  .  X  .
```

Eventually this table may be simplified to:

```
                        1  2
Credit limit exceeded   Y  ELSE
Prompt payer            N  .
Special clearance       N  .
Accept order            .  X
Reject order            X  .
```

3.2 Decision Tables in FILETAB

FILETAB decision tables allow program logic to be defined in a concise yet powerful manner; they allow files to be accessed, data to be printed, calculations and processing to be performed, and so on. Decision tables may be used to interrupt the Fixed Logic for specific purposes. For example, a value which is to be totalled or used as a control field may be computed; records may be selected for printing, control breaks may be forced and so on.

3.2.1 Facilities and Uses of Decision Tables in FILETAB

FILETAB processes limited, mixed and extended entry decision tables.

Any number of decision tables may be specified, and decision table linkage may be performed from conditions (via the `IF` verb), or actions (via the `CALL`, `EXIT` and `GOTO` verbs).

Various combinations of initial actions, conditions and actions may be specified.

Condition operators may be used to compare the value of a field with that of another field or with a constant (*see Condition Operators on page 16*).

The condition verb `LOOKUP` may be used to locate records on indexed files or entries in tables (*see Chapter 6*).

It is possible to add, subtract, multiply and divide.

Reading and writing of files and tables are performed by the verbs `READ` and `WRITE`.

An `ELSE` rule may be defined.

3.2.2 Interpretation of FILETAB Decision Tables

Conditions in FILETAB decision tables are first tested for relevance. If they are non-relevant, they are not evaluated. Non-relevant conditions are those which cannot change the outcome of the decision table. For example:

```
C  FIELD1 = 'A'     Y      Y      N
   FIELD2 = 'B'     N      Y      -
```

`FIELD2` is tested only if `FIELD1` is equal to 'A'.

Care should be taken with conditions which change the value of data items or the positions of Pointers (*see Chapter 8*). If these conditions are not evaluated, the data item or Pointer position remains unchanged.

Note: FILETAB may process decision tables in which more than one rule is satisfied, for example:

```
Sex    =     Male            Y      .      ELSE
Age    GT    21              .      Y      .

Spacefill cols. 1,2 and 3    X      X      X
Put * in column 1            X      .      .
Put * in column 2            .      X      .
Put * in column 3            .      .      X
```

Assuming that the data presented to the table consists of:

```
Name       Sex       Age
Edwards    Male       18
Thomas     Female     30
Jones      Female     16
Jenkins    Male       35
```

then the output produced is as follows:

```
Name       Column 1  Column 2  Column 3
EDWARDS    *
THOMAS               *
JONES                          *
JENKINS    *         *
```

Having evaluated the two conditions for 'Edwards', FILETAB finds that only rule 1 can be satisfied. For 'Thomas' and 'Jones', rule 2 and the ELSE rule are satisfied respectively. However, for 'Jenkins', after evaluating the two conditions, it is found that rules 1 and 2 are both satisfied. FILETAB always processes the actions by working serially through them, carrying out any actions which have at least one X specified in satisfied rules. So the first action, to spacefill the columns, is performed, then an * is moved to column 1, and an * is also moved to column 2. FILETAB decision tables are always interpreted in this way. For full details, *see Decision Table Techniques on page 34*

The following example indicates the way in which *Example 5 on page 6* might be written as a FILETAB decision table.

```
*DETAB    TEST05
C   RECTYPE    =  '05'      Y      Y      Y      ELSE
    QUANTITY  <  2500      Y      N      .      .
    VALUE      >  0         Y      .      N      .
A   GOTO    VALID          X      .      .      .
    GOTO    ERROR          .      X      X      .
    GOTO    OTHERS         .      .      .      X
```

RECTYPE, QUANTITY and VALUE were defined in a *DICTIONARY. VALID, ERROR and OTHERS are names of decision tables defined elsewhere. C and A introduce the first condition and action respectively. Exact rules for drawing up decision tables appear in *Decision Table Format on page 13*. Comparing this example with *Example 5 on page 6* shows that only minor changes are required to alter a decision table into FILETAB format.

Example 4 on page 5, written as a FILETAB decision table is:

```
*DETAB   UPDATE
C  TRANSKEY  >      MASTKEY    Y      N      N      N   ELSE
   TRANSKEY  =      MASTKEY    N      Y      Y      N      .
   TRANSKEY  <      MASTKEY    N      N      N      Y      .
   TRANSTYP  =      'I'        .      N      N      Y      .
   TRANSTYP  =      'D'        .      Y      N      N      .
   TRANSTYP  =      'A'        .      N      Y      N      .
A  WRITE  MASTOUT   MASTREC    X      .      .      .      .
   WRITE  MASTOUT   TRANSREC   .      .      X      X      .
   CALL   ERROR                .      .      .      .      X
   READ   MASTIN    MASTREC    X      X      X      .      .
   READ   TRANS     TRANSREC   .      X      X      X      X
   REPEAT                      X      X      X      X      X
```

In practice, an 'end-of-file' condition would have to be tested in this decision table, to provide an exit route.

Not all the condition tests here are needed since both for TRANSKEY and for TRANSTYP, satisfaction of the first condition makes the second and

third tests redundant, while satisfaction of the second test makes the third redundant. Irrelevant entries make the conditions more concise:

```
C  TRANSKEY  >    MASTKEY   Y      N      N      N      ELSE
   TRANSKEY  =    MASTKEY   .      Y      Y      N      .
   TRANSKEY  <    MASTKEY   .      .      .      Y      .
   TRANSTYP  =    'I'       .      .      .      Y      .
   TRANSTYP  =    'D'       .      Y      .      .      .
   TRANSTYP  =    'A'       .      .      Y      .      .
```

This is more meaningful, as well as more efficient. Alternatively, the conditions can be written, as in *Example 6 on page 6*, in extended entry form, as follows:

```
C  TRANSKEY    ?    MASTKEY   GT     =      =      LT    ELSE
   TRANSTYP    =     ?        .     'D'    'A'    'I'    .
```

3.2.3 Incomplete Decision Tables

An incomplete decision table is one in which not all possible combinations of conditions are catered for:

```
*DETAB TEST
C  RECTYPE       EQ  '20'     Y      Y      Y
   PRODCODE      EQ  '121'    Y      N      N
   PRODGP        EQ  '2'      .      Y      N
A  GOTO  TABA                 X      X      .
   GOTO  TABB                 .      .      X
```

This decision table is incomplete, because the rule for a `RECTYPE` not equal to 20 has been omitted. An incomplete decision table can always be completed by adding an `ELSE` rule.

The number of unique rules needed to complete a limited entry decision table is always equal to the number 2 raised to the power n, where n is the number of conditions. For instance, if 3 conditions are present, 8 different rules are required to complete the decision table (2 to the power 3 is 8).

A limited entry decision table should be checked for completeness before simplification, to ensure that no ambiguous rules are inadvertently counted. Ambiguous logic can result from a careless use of the irrelevance entry (dot or dash) in a condition. *See Decision Table Theory on page 1.*

For incomplete decision tables, FILETAB always provides an `ELSE` rule which is entered when no other rules are satisfied, and an `IGNORE` action for this implied `ELSE` rule. For example, the above table is interpreted as if it were:

```
*DETAB TEST
C  RECTYPE       EQ  '20'     Y      Y      Y      ELSE
   PRODCODE      EQ  '121'    Y      N      N      .
   PRODGP        EQ  '2'      .      Y      N      .
A  GOTO   TABA                X      X      .      .
   GOTO   TABB                .      .      X      .
   IGNORE                     .      .      .      X
```

The part supplied by FILETAB for an incomplete decision table is known as the 'implied ELSE/IGNORE rule'. Details of the effects of both implicit and explicit ELSE/IGNORE rules are given in *Chapter 4*.

3.3 Decision Table Format

This section describes the format and rules governing the use of decision tables in FILETAB.

3.3.1 The *DETAB Directive

Each decision table in a FILETAB program is introduced by the *DETAB directive and given a name. The format is:

```
*DETAB   detabname
```

detabname can be of unlimited length and may contain hyphens. It must however, begin with an alphabetic character. They may include a hyphen and must begin with an alphabetic character other than Z. Detabnames must be unique within a program. For example:

Valid detabnames	Invalid detabnames
TABLE1 FIRST A1	4A

The detabnames, START, RECORD, BREAK, ENDFILE and PRINT refer to user-defined decision tables which are entered automatically at the corresponding points in the Fixed Logic. These names must be only given to decision tables to be entered at these points, *see Chapter 4*.

3.3.2 Decision Table Detail Format

The *DETAB directive is followed by detail statements which define Initial Actions, Conditions and Actions, as required.

The first of a set of initial actions, conditions or actions must be introduced by I, C or A respectively in position 1, followed by at least one space.

When a decision table detail line is too long for a single 80-character FILETAB statement, the following line may be used as a continuation. Only one continuation line per detail statement is allowed.

Continuation is signalled to FILETAB by the presence of any non-space character in the last four character positions on the first line (that is, positions 77-80). The character (or characters) used can often be a valid part of the decision table detail statement.

Alternatively, the visible space character, represented by an exclamation mark (!) may be used to force continuation. Since ! is treated as a space

by the decision table analyser, it does not affect the syntax of the statement.

Note: ! is not treated as a space character when it appears as part of a character literal.

Care should be taken to ensure that no part of a detail line is inadvertently placed in positions 77-80, since this would cause the next detail line to be treated as a continuation incorrectly.

Note: This method of forcing continuation may be used only with decision table detail lines but see Chapter 2.1.7 for further details.

Comments may be included on a detail line. They may only follow the last condition or action entry on that line, and must be preceded by a left-hand square bracket, `[`. If any comment is present in positions 77-80, the following detail line is, of course, treated as a continuation.

```
*DETAB   READREC
I   READ  INFILE  BUFFER
C   BUFFER:4  =   -1       Y    N    [END OF FILE?
A   GOTO   END             X    .
    CALL  PROCESS          .    X
    GOTO  READREC          .    X
```

Only `I`, `C` and `A` may appear in position 1 of a detail line, except for one-of-a-set entries (*see Chapter 3.4*), and any continuation line. The following combinations of initial actions, conditions and actions are valid.

<table>
<tr><td>I
C
A</td><td>I
C</td><td>I</td><td>C
A</td><td>C</td><td>A</td></tr>
</table>

There is no limit to the number of details which may be specified as initial actions, conditions or actions, but no more than thirty one rules may be specified, including an explicit or implicit `ELSE` rule.

The basic format of a FILETAB decision table is:

Initial action stub	
Condition stub	Condition entry
Action stub	Action entry

In limited entry format, no entries are specified for initial actions. Items in the detail statements of decision tables must be separated by at least one space character.

3.3.3 Decision Table Stub Syntax - Limited Entry

Condition and action stubs may consist of operators or verbs. Condition and action operations have the format:

First operand		Second operand
*DICTIONARY name Field Definition *FSC Reserved Word Pointer Field Definition	Operator	*DICTIONARY name Field Definition *FSC Reserved Word numeric constant character literal hexadecimal literal Pointer Field Definition Address Constant Length Attribute

Note: A second operand is not always necessary.

Pointer Field Definition (PFD) and Address Constant and Length Attribute are described in *Chapter 8*. Condition and action verbs have the following format:

```
verb    filename      Field Definition
verb    tablename     Field Definition
verb    detabname
```

Some verbs do not require certain operands.

3.3.4 Condition Operators

EQ or =	equal to
LT or <	less than
GT or >	greater than
NE	not equal
GE	greater than or equal to
LE	less than or equal to
CT	contains
:	one-of-a-set

3.3.5 Condition Verbs

IF	enter another decision table, and, upon returning, test true or false resu
LOOKUP	access indexed files or *TABLEs to determine presence of a key

3.3.6 Action Operators

-	subtract
+	add
*	multiply
/	divide
S	spacefill
Z	zeroise
MV or :=	move
FL	fill
AND	logical AND
OR	logical OR
XOR	logical exclusive OR
SL	shift left
SR	shift right
ST	skip to
SP	skip past
SBT	skip back to
SBP	skip back past

3.3.7 Action Verbs

READ	read subsidiary file or *TABLE
WRITE	write to subsidiary file or *TABLE
CLOSE	close a file or *TABLE
SELECT	select the nth record from a *TABLE
DELETE	delete a record from an Indexed-Sequential or a *TABLE
GOTO	branch to a decision table
REPEAT	loop around the current decision table
CALL	branch to a decision table and return to instruction following CALL
EXIT	exit from a decision table entered by CALL or IF or the fixed logic
IGNORE	ignore the current record for printing
DISPLAY	display a character string
STOP	display a message and halt the program
INDEX	change the current index for an indexed sequential file with alternate indices.

Most of the verbs and operators listed above are described in *Decision Table Verbs and Operations on page 21*, although the descriptions of verbs related to files and *TABLE appear in *Chapter 6*.

The following are definitions of previously used terms:

*DICTIONARY name	any name previously defined in a *DICTIONARY.
Field Definition	a valid Field Definition with one of the formats as describ in *Chapter 2* and *Chapter 8*. for example, 10:4, -50, 10, REC+16:8.
*FSC	* followed by any Field Specifying Character, defined in *INLIST for example, *M, *A.
Reserved Word	any FILETAB Reserved Word for example, COUNT.
numeric constant	number, optionally preceded by + or - with up to 19 digi for example, 2, -345
hexadecimal constant	a string of from 1 to16 hexadecimal digits defining a constant, for example, X'001234' Unless all 16 digits ar specified, it is treated as an unsigned value.
character literal	any character string from 1 to 70 characters in length, enclosed in single quotes. It can contain ? only in extend format, for example, 'SMITH', '****', 'AA22', '60'
hexadecimal literal	a string of from 2 to 80 hexadecimal digits; it is treated a character string, for example, X'616263'. *Note: For Intel (where Byte order is LSB[0] to MSB[n]) Hex literal values are in the order specified. If used to s numeric values the order should be reversed for Intel machines.* *For example, 0:2 MV X'0001' places 256 in 0:2 on Intel and 1 on Sparc, 0:2 MV X'0100' places 1 in 0 on Intel and 256 on Sparc and 0/2 MV X'415A' pla 'AZ' in 0/2 on both machines* The use of hex literals to set numeric fields is NON PORTABLE
Pointer Field Definition	a special form of Field Definition incorporating an indire address., for example, (P1):4,(P2)P4 (see *Chapter*
address constant	A' followed by any *DICTIONARY name, Reserved Wo *FSC or start-position (see *Chapter 8*)
length attribute	L' followed by any *DICTIONARY name, Reserved Wo *FSC or valid field definition (see *Chapter 8*)

localname	a localname as defined on `*FILE`, `*IFILE`, `*OFILE`, `*IOFILE` (see *Chapter 6*)
tablename	the name of a user-defined `*TABLE`
detabname	the name of a user-defined `*DETAB`
Operator	any FILETAB operator, for example, `EQ, +, MV`
Verb	any FILETAB verb, for example, `IF, READ`

3.3.8 Decision Table Entry Syntax - Limited Entry

The following items may appear in the condition entry of a limited entry FILETAB decision table.

`Y`	for	Yes
`N`	for	No
`. or -`	for	irrelevant
`ELSE`	for	the `ELSE` rule

`ELSE` may appear only once in a decision table, and it must be in the last rule of the first condition. Only dot or dash may appear in this rule for the remaining conditions.

The items allowed in the action entry of a limited entry FILETAB decision table are:

`X`	for	perform the action
`. or -`	for	do not perform the action

If the whole entry is blank, FILETAB assumes that an `X` is required for every rule stated. This technique is not recommended, since it makes decision tables difficult to read, and program logic difficult to follow.

3.3.8.1 Example

```
*DETAB  TEST
I   WORK   MV   QTY
    WORK   *    COST
C   PRODCODE    =      '199'  Y   Y   N   ELSE
    PRODCODE    =      '200'  .   .   Y   .
    WORKGT      GT     1000   Y   N   .   .
A   CALL   PROCESS            X   X   X   X
    GOTO   SPECIAL            X   .   X   .
    GOTO   ORDINARY           .   X   .   X
```

3.3.9 Decision Table Stub Syntax - Extended Entry

In an extended entry table, any operand or part of an operand in the stub may be represented by `?`, and the missing part is specified in the entry. Only one question mark may be specified in a line.

3.3.10 Decision Table Entry Syntax - Extended Entry

In an extended entry table, the entry portion consists of the items represented by `?` in the stub. A dot or dash is used to indicate 'irrelevant' in the condition entry and 'do not perform the action' in the action entry. In the following examples, the bracketed lines are equivalent:

```
 PRODCODE   =   ?   100    120   140
 PRODCODE   =   1?   00     20    40
 PRODCODE   =   1?0   0      2     4

 ?      =   WORK1  -8/4  -20/4
 ?/4    =   WORK1  -8      -20
 -?/4   =   WORK1   8       20

 GOTO ?       TABLE1  TABLE2  TABLE3
 GOTO TABLE?  1       2       3
```

When the entry portion of a decision table line consists entirely of alphanumeric (`A-Z`, `0-9`) literals or irrelevance (`.` or `-`) entries (and optionally an `ELSE`), an alternative format is available. Instead of surrounding each literal with quotes, the quotes may surround the question mark. For example, the following two lines are equivalent:

```
WRKA  =  ?       'AB'   'CD'   .   'XY'   ELSE
WRKA  = '?'      AB     CD    .   XY    ELSE
```

Further examples of extended entry are given in Chapter 3.4.

3.3.11 Decision Table Initial Action - Extended Entry

If an extended entry is specified within a group of initial actions, it is converted into the equivalent limited entry, and each of the resulting actions is obeyed. For example:

```
*DETAB INIT
I ? Z       WORK  WORK1  WORK2
```

This is equivalent to:

```
*DETAB INIT
I WORK  Z
  WORK1 Z
  WORK2 Z
```

Note: This applies equally to extended entries in action-only decision tables.

3.4 Decision Table Verbs and Operations

3.4.1 Introduction

This section describes the rules for the use of the verbs and operations listed in Chapter 3.3, with examples. File and table handling verbs are described in detail in *Chapter 6*.

Note: In decision table operations, Address Constants and Length Attributes (see Chapter 8*) are treated as numeric constants. Hexadecimal values are treated as constants if the operand on the left is binary, and as character literals if the operand on the left is character.*

3.4.2 Comparison Operators

When a character field is compared with another character field, or with a character literal, the length is determined by the field on the left. If the lengths are unequal, the field on the right is either truncated from the right, or extended by the addition of trailing spaces, as appropriate.

When either, or both, of the fields is a binary or a numeric constant, a numeric comparison takes place. The length is taken from the longer operand.

3.4.3 One-of-a-set

Where a field is to be compared with a range of values, the one-of-a-set test may be used:

```
field  operator  'number of entries'delimiter
```

followed by the entries on the subsequent detail line.

`field` is a signed binary (`:4` or `:8`) or a character field.

`operator` is a colon (`:`).

’`number of entries’ delimiter` defines the number of entries in the set, separated and terminated by the specified delimiter. The entries may contain any characters except the delimiter, which must not be a space, a left-hand square bracket (`[`) or an exclamation mark (`!`). Alphanumeric values must not be enclosed within quotes. Apostrophes are taken as part of the value. No individual entry may be longer than eighty characters (including the delimiter). The usual padding and truncation rules apply.

The entries need not be specified in any special order, but they must begin in position 1 of the detail statement, for example:

```
C FIELD1   :   5,          Y  N
1,3,5,7,9,

C 100/3    :   6*          Y  N
ABC*CDE*AAA*BBB*ZZZ*123*
```

In the example, if FIELD1 equals any of the values 1, 3, 5, 7 and 9, the condition returns TRUE (YES); otherwise, it returns FALSE.

Note: Care should be used if writing One-of-a-set in extended entry form.

*TABLE and the LOOKUP verb may be used instead of one-of-a-set. For details, see *Chapter 6.*

*Note: One-of-a-set uses *TABLE and LOOKUP. If a field contains high values, that is, X'FF', then one-of-a-set returns TRUE due to the presence of the 'end-of-table-entry'.*

The example would also return TRUE if FIELD1 contained -1 (for example, X'FFFFFFFF').

3.4.4 Arithmetic Operations

The fields on either side of an arithmetic operator may be character or binary; the field on the right may be a numeric constant, a hexadecimal constant, a character literal, a Length Attribute or an Address Constant. The result is always stored in the field on the left. The field must be defined as big enough to hold the result.

The following arithmetic operators are used:

+	add
-	subtract
*	multiply
/	divide

```
*DETAB   CALC
A -8:4       *     100   [ MULTIPLY BY 100
  -8/4       -     60:4  [ SUBTRACT CONTENTS OF 60:4
  -20:2      +     1     [ ADD 1
```

Note: It should be noted that FILETAB can process only integer numbers. This is particularly important in division operations, where the result (quotient) is always an integer value. However, the remainder from any division is stored in the Reserved Word REMAINS (see Chapter 7.1).

FILETAB also provides either a rounded or an unrounded quotient, as requested on the *OPTION directive (see ** OPTIONs on page 1 of Chapter 5*). For example, if the field POS contains the value 8, and the field NEG contains -8, the following results are obtained from the divide operator (/)

.

Divide unrounded	Result after division	REMAINS
POS / 3 POS / -3 NEG / 3 NEG / -3	2 -2 -2 2	2 2 -2 -2
POS / 7 POS / 3 POS / -3 NEG / 3 NEG / -3	1 3 -3 -3 3	1 -1 -1 1 1

In all cases, the dividend equals the quotient multiplied by the divisor, plus the contents of `REMAINS`.

3.4.5 Spacefill and Zeroise Operations

Spacefill (`S`) is used to fill a character field with spaces. Zeroise (`Z`) is used to fill a field with zeros.

```
*DETAB       INITIAL
A  -60/40    S
   -8/8      Z
   -10:2     Z
   TOTAL     Z
```

When `Z` is used, the actual bit pattern contained in the operand is dependent on the type of the field, that is:

four-byte character zero four-byte binary zero	= `X'30303030'` in hexadecimal = `X'00000000'` in hexadecimal

3.4.6 Fill Operation

The `FL` operation allows character fields, character literals and hexadecimal literals to be moved to a character field.

The receiving field is on the left and determines the number of bytes moved. The receiving field is filled from the left. If it is shorter than the sending field, the excess characters on the right of the sending field are not transferred. It it is longer, the sending field is repeated until the required number of bytes has been transferred. For example:

`0/80      FL 'A'`
`200/200   FL X'00'`
`1000/100  FL 100/3`
	fills 0/80 with `A`s
	zeroises 25 `:8` fields beginning at byte 200
	repeatedly fills `1000/100` with the 3 bytes held at `10(`

3.4.7 Logical Operations

FILETAB can perform the following logical operations: AND, OR, EXCLUSIVE OR, SHIFT LEFT and SHIFT RIGHT. Formats and further details follow.

3.4.7.1 AND, OR and EXCLUSIVE OR

```
field1      logical operator      field2
```

If field1 is a character field (/), field2 may be:

	Examples
1 A character field 2 A character literal 3 A hexadecimal literal 4 The *DICTIONARY name of a character field	100/3, 100/4 'ABCD', '1X2Y3Z' X'60616263' CHARFLD

If field1 is a binary field (:), field2 may be:

	Examples
1 A binary field 2 A numeric constant 3 A hexadecimal constant 4 The *DICTIONARY name of a binary field	200:4, 182:2 -500, 1234 X'00003521' ERIC1

3.4.7.2 Logical AND

The operator for this instruction is AND. The logical operation matches the corresponding bits in the two operands, and the result is placed in field1. The action of the AND operation is best illustrated by the following table.

First operand	Second operand	Result
0 0 1 1	0 1 0 1	0 0 0 1

3.4.7.3 Logical OR

The operator for this instruction is `OR`. The logical operation matches the corresponding bits in the two operands, and the result is placed in `field1`. The action of the `OR` operation is best illustrated by the following table.

First operand	Second operand	Result
0 0 1 1	0 1 0 1	0 1 1 1

3.4.7.4 Logical EXCLUSIVE OR

The operator is `XOR`. The `XOR` instruction matches the corresponding bits in the two operands, and puts the result in `field1`. `XOR` is best illustrated by the following table.

First operand	Second operand	Result
0 0 1 1	0 1 0 1	0 1 1 0

3.4.7.5 SHIFT LEFT and SHIFT RIGHT

The format for these instructions is:

```
field1      logical operator field2
```

`field1` must be a binary field of length 4 or 8.

`field2` shows how many bits are to be shifted, and must be a numeric or hexadecimal constant, or a binary field.

3.4.7.6 Logical Shifts

The operators are `SL` and `SR` (Shift Left and Shift Right). Any bits displaced by the shift are lost. The sign bit is not propagated. The bits inserted at the opposite end of the field to those displaced are set to zero.

Shift Left (SL) is equivalent to multiplying `field1` by 2 to the power represented by `field2`.

Shift Right (SR) is equivalent to dividing `field1` by 2 to the power represented by `field2`.

Example:

The :4 field VALUE holds 23

```
VALUE SL 2  [2 to the power 2 = 4
```

VALUE now holds 92

```
VALUE SR 6 [2 to the power 6 = 64
```

VALUE now holds 1

3.4.8 Move Operation MV

The MV operation allows fields to be moved within and between the Record, Work, *INLIST and Print Areas, Reserved Words to be accessed, records for writing to output files to be formatted and convertion from one data type to another.

The receiving field is the field on the left; this field also determines the type of move operation and the number of bytes moved.

In a move, the contents of the receiving field are over-written by the contents of the sending field. The contents of the sending field remain unaltered, for example:

```
FIELDB  MV  FIELDA
```

The contents of FIELDA are moved to FIELDB. FIELDA remains unaltered.

When the fields involved in a move are of the same type and length, the receiving field becomes an exact copy of the sending field.

In the case of character to character moves, care should be taken when the sending and receiving fields overlap. If the receiving field is to the left of the sending field, the data is moved to the left as expected. However, if the receiving field is to the right of the sending field, the results are unpredictable.

3.4.9 Moves of Similar Field Types

When the sending and receiving fields are of the same type, but of different lengths, the following rules apply:

3.4.9.1 Character Moves

The receiving field is always filled from the left. If it is shorter than the sending field, the excess characters on the right of the sending field are not transferred. If it is longer, the field is padded with spaces to the right. Care must always be taken when the fields involved overlap.

3.4.9.2 Binary Moves

Values may be transferred freely between binary fields provided that the value in the sending field does not exceed the capacity of the receiving field. If it does, a program error occurs. Care should be taken when signed to unsigned binary moves are performed. If the sending field is negative, a program error occurs. It should also be remembered that a larger positive number can be held in an unsigned field than in a signed field of the same length.

3.4.10 Moves of Dissimilar Field Types

3.4.10.1 Character to Binary Moves

The characters are converted to binary. Care should be taken that the character field contains only numeric characters, as non-numeric characters are converted without an error being signalled. Negative signs are ignored; positive signs give incorrect results. Overflow of the receiving field causes a program error. Because negative signs are not transferred, the resulting binary field always contains a positive or zero value.

Note: Spaces are converted to zero. For example:

```
A 6/6  MV '100'
  12:6 MV  6/6
```

gives a value of `100000` in `12:6`.

3.4.10.2 Binary to Character Moves

The binary number is converted, and moved into the character field from the right. Any unused characters in the receiving field are filled with spaces or zeros, depending on the editing `*OPTION` (*see Chapter 5*) in force. If the binary field is signed, the sign is ignored, so the resulting character field always contains a positive or zero value. A program error occurs if the character field cannot hold the converted binary number.

3.4.10.3 Hexadecimal Literal to Character Moves

No conversion is necessary, because the bit patterns of hexadecimal pairs are identical to those of the corresponding characters.

The receiving field is always filled from the left. If it is shorter than the sending field, the excess characters on the right of the sending field are not transferred. If it is longer, the field is padded with spaces to the right.

3.4.10.4 Hexadecimal Constant to Binary

The rules for moving hexadecimal constants into binary fields are identical to those for moving binary fields into binary, with one proviso: only 16-digit hexadecimal constants are treated as signed.

3.4.11 Multiple Move Operation MV

The operator is identical to the one for simple moves. The only difference is in the operands, which take the form:

```
CFIELDX,CFIELDY...etc   MV   CFIELDA,CFIELDB...etc
```

The receiving or sending operands may be one single field or multiple fields. Receiving fields must be character fields. Sending fields must be either character fields or character literals, or a mixture of both. Where receiving/sending fields are multiple fields, they need not be contiguous.

3.4.12 Decision Table Linking

FILETAB supports a number of verbs which link decision tables or perform recursion. Decision tables may be joined by the `GOTO` verb, or by the `CALL`, or `IF` verbs which allow subroutines to be entered. Exit from a decision table entered in this way occurs automatically when there are no more actions to follow. Controlled exit can be achieved by using the `EXIT` verb. The `EXIT` verb can return a True (`EXIT T`) or False (`EXIT F`) parameter to the previous decision table. If the parameter is not explicitly supplied a default is returned. Recursion is achieved by using `GOTO` the current decision table name or `REPEAT`.

3.4.12.1 Action Verb GOTO

When `GOTO` is obeyed in a decision table action, it transfers control to the first detail line in the named decision table.

```
GOTO   detabname
```

Note: `detabname` may be the name of the current decision table, in which case control is transferred to the first statement in the current table.

When `GOTO same table` is used, one of the rules must provide for an exit from the decision table, otherwise a loop occurs.

An example of the use of `GOTO` follows.

```
*DETAB  FIRST
C  F1  =  F2       Y     N
A  GOTO THIRD      .     X
   -4:4    MV  4   X     .
   GOTO SECOND     X     .
*
*DETAB  SECOND
C  -4:4    =  0    Y     N
A  -4:4    -  1    .     X
   F3  *  10       .     X
   GOTO SECOND     .     X
   GOTO THIRD      X     .
*
*DETAB  THIRD
I  F1  *  F3
C  F1  GT  F2      Y     N
A  GOTO  THIRD     .     X
   GOTO  NEXT      X     .
```

`GOTO` may be used in extended entry decision tables thus:

```
C  RECTYPE   =  '?'   05       06       07
A  GOTO      ?        TABLE5   TABLE6   TABLE7
```

or alternatively:

```
C  RECTYPE   =  '0?'  5       6        7
A  GOTO TABLE?        5       6        7
```

When all relevant actions have been performed in a decision table entered by `GOTO`, control returns to the last `CALL`ing decision table, or to the FILETAB Fixed Logic. It does not return to the decision table that contained the `GOTO`.

3.4.12.2 Action Verb REPEAT

```
REPEAT  [no operands
```

When REPEAT is obeyed, processing is returned to the first condition of the current decision table. REPEAT must not be used in decision tables without condition entries.

REPEAT and GOTO same table behave differently only when the decision table contains initial actions. REPEAT does not perform the initial actions. GOTO same table, however, performs them on each re-entry of the decision table. The following examples, both of which serve the same function, indicate this difference.

Note: Because of FILETAB's internal operations, REPEAT is the more efficient.

```
*DETAB LOOP
I  READ    INP   BUFF/8
C  BUFF/3  EQ    'END'      N    ELSE
A  TOT     +     BUFF+4:4   X    -
   READ    INP   BUFF/8     X    -
   REPEAT                   X    -
*DETAB LOOP1
I  READ    INP   BUFF/8
C  BUFF/3  EQ    'END'      N    ELSE
A  TOT     +     BUFF+4:4   X    -
   GOTO    LOOP1            X    -
```

3.4.12.3 Action Verbs CALL, EXIT

```
CALL   detabname
EXIT   T
EXIT   F
EXIT      [no operand
```

CALL is similar in effect to GOTO, in that it transfers control to the first line in the decision table named as the operand of the CALL. When an EXIT verb is obeyed, control is passed back to the instruction following the CALL.

It is possible to nest CALL verbs (that is, a CALLed table may CALL another).

EXIT need not appear in the CALLed table, and other tables may be linked by GOTO verbs before the EXIT is obeyed.

If a decision table entered by a CALL verb does not contain an EXIT or a GOTO, EXIT is supplied as the last action.

The CALL verb may be written in extended entry form:

```
C  RECTYPE  = '?'      05       06       07
A  CALL     ?          TABLE5   TABLE6   TABLE7
```

or alternatively:

```
C  RECTYPE = '0?'      5        6        7
A  CALL   TABLE?       5        6        7
```

3.4.12.4 Condition Verb IF, Action Verb EXIT

```
IF      detabname
EXIT    T
EXIT    F
EXIT       [no operand
```

`IF` is similar to `CALL`. `IF` transfers control to the specified decision table. An `EXIT T` (true) or `EXIT F` (false) or `EXIT` (default 'true') returns control to the original table. An `EXIT T` or `EXIT` returns a `YES` answer to the `IF` condition, and an `EXIT F` returns a `NO` answer.

```
*DETAB  TEST
C RECTYPE  =  '05'         Y   Y   ELSE
  IF CONDS                 Y   N   .
A GOTO READNEXT            .   X   X
  GOTO PRIN                X   .   .
*DETAB CONDS
C CLOSEIND =  0            Y   ELSE
  OUTSBAL  GT CRCODE       Y   .
A EXIT T                   X   .         [RETURNS Y TO IF
  EXIT F                   .   X         [RETURNS N TO IF
```

In `*DETAB TEST`, if `RECTYPE` is `05`, then rule 1 or rule 2 is satisfied. Rule 1 is satisfied if `*DETAB CONDS` returns a 'True' answer to `*DETAB TEST` (that is, `CLOSEIND` is `0`, and `OUTSBAL` is greater than `CRCODE`); if a 'False' answer is returned, rule 2 of `*DETAB TEST` is satisfied.

The `IF` verb may be written in extended entry form, but, if it is, only `EXIT T` can be tested for. For example:

```
C   IF  ?       ONE    TWO
```

This is interpreted as:

```
C   IF   ONE     Y      .
    IF   TWO     .      Y
```

If no `EXIT T`, `EXIT F` or `EXIT` is specified in a decision table entered as a result of an `IF`, then an `EXIT T` is inserted by FILETAB for the rules specified. If the decision table entered as a result of an `IF` is incomplete, FILETAB provides an implicit `ELSE` rule with an `IGNORE` action. An `IGNORE` in an `IF`ed table is interpreted as `EXIT F`.

```
*DETAB  CALC
C  IF  RANGE      Y   N
etc.
*DETAB  RANGE
C  AREA  GE  '?' 00   30
   AREA  LE  '?' 19   49
```

If `AREA` is in range `00-19` or `30-49`, `*DETAB RANGE` exits true; if not, it exits false, because the table is interpreted as:

```
*DETAB  RANGE
C  AREA   GE  '?'   00   30   ELSE
   AREA   LE  '?'   19   49   .
A  EXIT       ?     T    T    F
```

3.4.12.5 Action Verb IGNORE

`IGNORE`

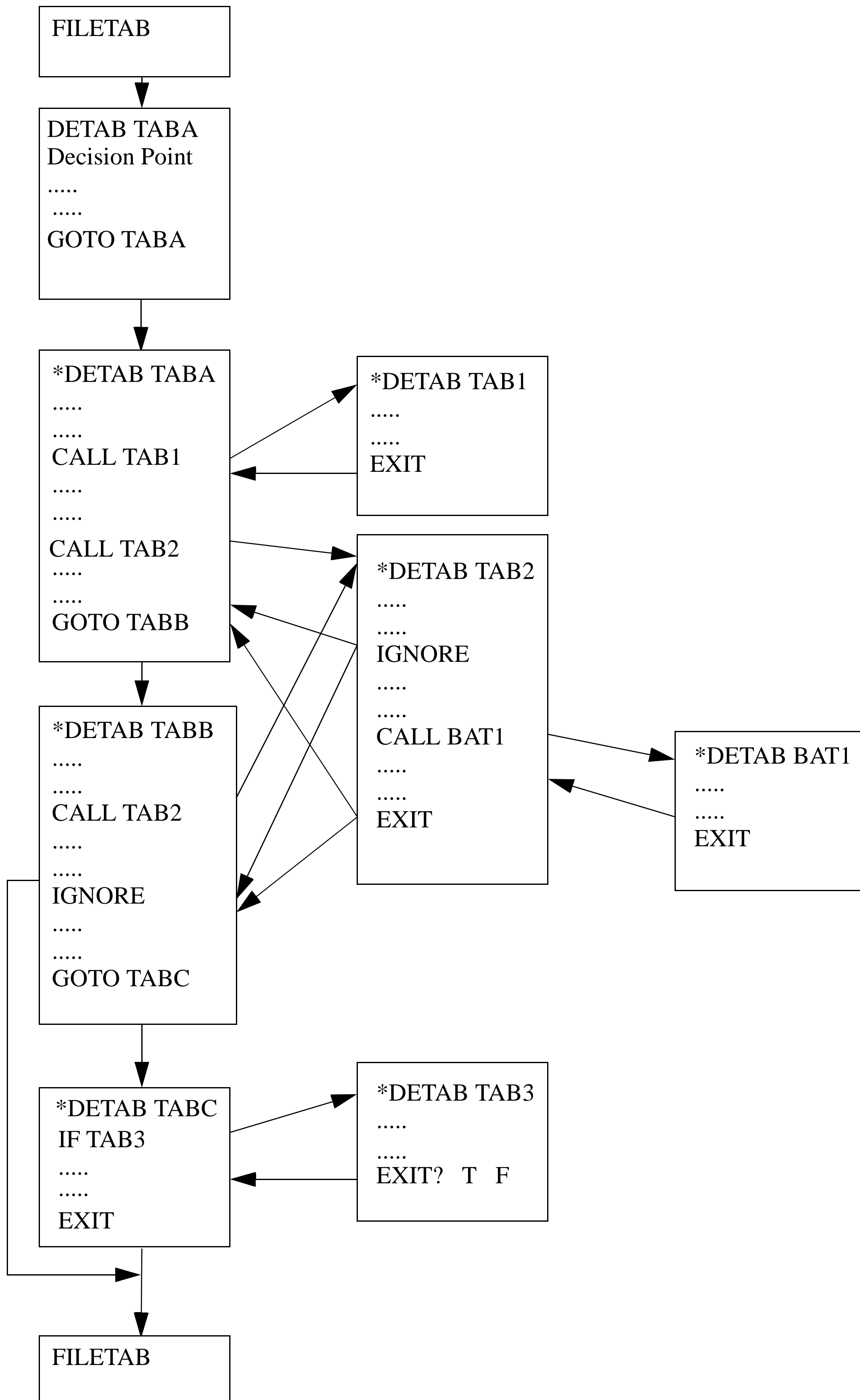
This verb is used to ignore a record; its effects are described fully in the sections devoted to Decision Points in *Chapter 4*.

3.4.12.6 Note on Linkage Diagram

The linkage diagram which follows on the next page is designed to illustrate the various possibilities which are allowed by the verbs dealt with so far - `GOTO, IF, CALL, EXIT` and `IGNORE`.

The `REPEAT` verb is not included in this diagram since its effect is confined to the decision table within which it appears.

3.4.12.7 Linkage Diagram



3.4.12.8 Action Verb DISPLAY

DISPLAY, outputs the contents character field, 'character literal or hexadecimal literal', for example:

```
DISPLAY   CHARFLD
DISPLAY   100/10
DISPLAY   'START OF PROGRAM ABC'
DISPLAY    X'313233'
```

where CHARFLD is defined in the *DICTIONARY as a character field (/).

3.4.12.9 Action Verb STOP

```
STOP   binary field
STOP   numeric constant
```

When a STOP verb is encountered, the program stops, for example:

```
STOP   STOPCODE
STOP   100:4
STOP   -1234
STOP   62
```

3.4.12.10 Other Action Verbs

A number of other action verbs can be used in FILETAB programs. These relate to the reading and writing of records on files, and entries in *TABLEs, and are described in *Tables on page 24 of Chapter 6*.

3.5 Decision Table and Fixed Tabulating Logic

The flow diagram in *Fixed Logic on page 6 of Chapter 2* summarises the Fixed Logic used when FILETAB is producing a simple report with no data selection or processing. The decision tables in FILETAB allow this Fixed Logic to be interrupted at various points and user-defined decision tables, which specify data selection and processing, to be entered. Most of the decision table operations available are described in *Decision Table Verbs and Operations on page 21*.

FILETAB can be set to enter decision tables through reserved decision table names. These are the names of specific decision points at which tables are to be entered:

`*DETAB START`	entered at the start of the run before the main file is opened.
`*DETAB RECORD`	entered each time a record has been read from the main file.
`*DETAB BREAK x`	entered at the end of a control group (or groups).
`*DETAB BREAKH x`	entered immediately prior to a control break heading (`BREAK` Decision Point).
`*DETAB ENDFILE`	entered at the end of the main file.
`*DETAB PRINT`	entered each time a line is ready to be printed.

More detailed notes on these Decision Points, and on their interface with the FILETAB Fixed Logic, may be found in *START Decision Point on page 9 of Chapter 4*.

3.6 Decision Table Techniques

This section examines techniques employed by FILETAB in its processing of decision tables.

3.6.1 Rule Mask Technique

Note: It is only necessary to read this section if a detailed understanding of FILETAB decision table implementation is required.

The rule mask technique relates to the handling of limited entry decision tables. FILETAB converts all extended and mixed entry decision tables to limited entry format during its translation phase. A description of this conversion technique appears in *Conversion of Decision Tables to Limited Entry Format on page 35*.

(For the sake of brevity, the following analysis of a limited entry decision table does not mention the `ELSE` rule.)

Every condition in a decision table has associated with it a relevance mask. This mask has a bit set (1) for Yes and No condition entries, and clear bits (0) for all irrelevant entries. Before each condition is evaluated, its relevance mask is tested to ascertain whether the result of the condition affects the rule or rules being obeyed. For instance:

```
*DETAB ONE                         [relevance mask
C  FIELD1    =   'A'   Y  Y  N    [    111
   FIELD2    =   'B'   Y  N  -    [    110
```

FIELD2 would be tested only if FIELD1 were equal to A. Care should be taken that the correct path is followed whenever un-evaluated conditions may affect a Pointer value or the contents of a data field.

In addition to the relevance mask, each condition has a false mask. This reflects, in a binary form, the No condition entries. By using logical instructions on the two masks, FILETAB can determine all true, false or irrelevant condition entries.

As each condition is evaluated, a running rule mask is maintained. This has bits set which correspond to the applicable rule or rules.

Each action entry has an associated action mask. This has bits set for each X action entry. Logically ANDing the action mask with the running rule mask determines whether the action is performed. The action is performed only if the result of the AND is non-zero.

Where consecutive lines have identical action entries, the specified actions are combined, and only one action mask is produced.

Although several rules may be satisfied, specific actions are performed only once.

All actions are performed sequentially, regardless of which rules are satisfied. For instance:

```
*DETAB ONE
C  FIELD1  =   'A'     Y  -  N
   FIELD2  =   'B'     -  Y  N
A  FIELD3  MV  '1'     .  X  .
   FIELD3  MV  '2'     X  .  .
   GOTO TWO            X  X  .
   GOTO THREE          .  .  X
```

If rules 1 and 2 are both true (that is, FIELD1 equals 'A' and FIELD2 equals 'B'), then the two MV actions are performed before the GOTO, so that FIELD3 contains '2' when *DETAB TWO is entered.

3.6.2 Conversion of Decision Tables to Limited Entry Format

Detail lines in FILETAB decision tables allow two different syntactic constructions. The first type, limited entry, relies upon decisions of a 'yes/no' nature within conditions, and 'do/don't' in actions. Extended entry format permits much more flexibility of expression. The two decision tables shown below are identical in meaning although they are very different in appearance.

```
*DETAB FIG1A                *DETAB FIG1B
C FLAG EQ 1   Y   .   .     C FLAG EQ ?   1    2    3
  FLAG EQ 2   .   Y   .     A GOTO ?      TA   TA   TB
  FLAG EQ 3   .   .   Y
A GOTO TA     X   X   .
  GOTO TB     .   .   X
```

Extended entry constructions use a question mark to expand the scope of conditions and/or actions. Rules are still effectively vertical, as in limited

entry, with the symbols in each rule being substituted for the question mark during translation of the decision table.

From this simple example, it is evident that a major advantage of extended entry format is that the size of the table is reduced, whilst the documentary aspect is retained.

As far as the translation of extended entry decision tables is concerned, FILETAB converts all such coding into a limited entry structure by a process of symbol substitution. Substitution occurs before the logical acceptability of a line is checked. The two decision tables, `FIG2A` and `FIG2B`, show the effects of symbol substitution. `FIG2B` is the limited entry equivalent of `FIG2A`.

```
*DETAB FIG2A                      *DETAB FIG2B
C FLDA ?  FLDB    LT EQ GT        C FLDA LT FLDB   Y . .
A IND  MV ?       0  1  2           FLDA EQ FLDB   . Y .
  CALL TAB?       LO EQ HI          FLDA GT FLDB   . . Y
  -?:4 +  1       4  4  8         A IND  MV 0      X . .
                                    IND  MV 1      . X .
                                    IND  MV 2      . . X
                                    CALL TABLO     X . .
                                    CALL TABEQ     . X .
                                    CALL TABHI     . . X
                                    -4:4 +  1      X X .
                                    -8:4 +  1      . . X
```

As in limited entry format, irrelevance or unrequired action must be indicated. In FILETAB decision tables, both dot and dash denote this. A dash however is open to confusion, since it could be interpreted as a minus-sign during symbol substitution. To remove any possibility of ambiguity, a dot should always be used in preference to a dash in the condition and action entries of decision tables.

Since FILETAB uses a Rule Mask Technique to interpret its decision tables, the maximum number of rules which can be handled is determined by the size of the masks. FILETAB reserves one word for each mask. The maximum number of rules in a decision table is thirty one, including the implicit or explicit `ELSE` rule.

In extended entry decision tables, the greater likelihood of conditions not catering for all possibilities increases the significance of an `ELSE` rule.

The inclusion of an `ELSE` as the last rule of the first condition in a decision table serves as a collector for the whole table when none of the other rules is applicable to particular data. Space must be reserved in the masks for the `ELSE` rule, and this occupies one bit of the word. However, as in all FILETAB decision tables, the absence of an explicit `ELSE` rule results in the implicit `ELSE` rule (action `IGNORE`) being obeyed when necessary.

Because lines of extended entry logic are converted internally into limited entry format, such logic cannot be processed any more rapidly than when expressed in limited entry form. The important economies are those of less bulky coding, flexibility of expression and ease of updating.

Two features of extended entry decision tables require at least a basic understanding of FILETAB's processing techniques.

Firstly, all conditions in a decision table are evaluated before any actions can be performed. Conditions which include the following must be used carefully, since they imply actions during their evaluation:

- `IF` (which links to another decision table where actions may be obeyed)
- `LOOKUP` (which positions an indicator within a table or an indexed file)
- Pointer operations (see *Chapter 8*)

Secondly, because of the method by which extended entry decision tables are expanded, situations where more than one rule is satisfied can have undesirable consequences. For instance, in the example below, decision table `FIG3A` is expanded into `FIG3B`. If rules 2 and 3 are both satisfied, the action `COUNTER` + 1 is obeyed once only.

```
*DETAB FIG3A                *DETAB FIG3B
C AMOUNT  GT ?   30 20 10   C AMOUNT  GT 30   Y . .
A COUNTER +  ?    .  1  1     AMOUNT  GT 20   . Y .
                              AMOUNT  GT 10   . . Y
                            A COUNTER +  1    . X X
```

The expansion of the extended entry action recognises the fact that rules 2 and 3 require an identical action to be performed.

```
*DETAB FIG3C                *DETAB FIG3D
C AMOUNT  GT ?   30 20 10   C AMOUNT  GT 30   Y . .
A COUNTER +  ?    .  1  01    AMOUNT  GT 20   . Y .
                              AMOUNT  GT 10   . . Y
                            A COUNTER +  1    . X .
                              COUNTER +  01   . . X
```

When we examine decision table `FIG3C`, the extended entry action appears semantically identical to `FIG3A`, but when FILETAB expands the coding, it is different, because the actions in rules 2 and 3 are not lexically identical. Decision table `FIG3D` illustrates the fact that in similar circumstances, `COUNTER` would be updated once more than in table `FIG3B`.

It should also be noted that the check for lexically identical entries is terminated as soon as a difference is found. For example:

```
*DETAB FIG3E                  *DETAB FIG3F
C AMOUNT  GT ?  50 40 30 20   C AMOUNT  GT 50   Y . . .
A COUNTER +  ?   .  1  2  1     AMOUNT  GT 40   . Y . .
                                AMOUNT  GT 30   . . Y .
                                AMOUNT  GT 20   . . . Y
                              A COUNTER +  1    . X . .
                                COUNTER +  2    . . X .
                                COUNTER +  1    . . . X
```

Here, the action `COUNTER` + 1 has been split into two entries, because the search for lexically identical entries has been terminated by the entry `'2'` in the third rule.

In conclusion, it is worth noting that extended entry notation is a powerful tool, and should always be used with due consideration of the way in which FILETAB handles such structures.

3.7 Decision Tables Summary

The decision table feature of FILETAB provides powerful processing instructions with which to:

- Specify program logic
- Select records for processing and/or printing
- Create, read and update subsidiary files
- Perform processing and calculations on records and work fields
- Print data on the line-printer
- Manipulate tabular data

Examples of reports and decision tables appear in *Chapter 11*.

3.7.1 Decision Table Terminology

```
*DETAB NAME
```

Initial action stub	
I WORK MV 100 WKA Z WKB Z	
Condition stub	**Condition entries**
C AREA = '05' WKA = ? WKB = ?	Y Y ELSE 0 1 . 1 1 .
Action stub	**Action entries**
A WKA + 1 WKB MV INDIC GOTO TABA GOTO NEXT	X . . . X X . . X . X .
	1 2 3
	Rules

3.8 Worked Example

This example is an extension of the program shown in *Worked Example on page 29 of Chapter 2*. The program now performs record selection.

A report is produced showing Area, Account Number and Balance for customers whose details are held on the customer master file. Customers with Account Numbers in the range 9000-9799 are to be excluded. Balance totals appear for each Area and at the end. Records are to be printed in the sequence of Account Number within Area.

```
*PROGRAM EXAMPLE
*FILE CUSTFILE RL=332  NAME=custfile.txt
```

```
*DICTIONARY
AREA     = 2/2           [ AREA
ACCNO    = 4/4           [ ACCOUNT NUMBER
BALANCE  = 324:4         [ BALANCE
*INLIST
M AREA
A ACCNO
2 BALANCE
*HEAD L CH1,2,55
DD/MM/YY     CUSTOMER BALANCE LISTING     PAGE PPPP



   AREA                  ACCOUNT        BALANCE

                         NUMBER

*OUT L 1,0
     MM                     AAAA      £22222.22
*OUT M 2,1
    'AREA'MM'TOTAL'                  £222222.22
*OUT F 3,0
    'FINAL TOTAL'                   £2222222.22
*DETAB RECORD
C ACCNO GE '9000'       Y   ELSE    [ IGNORE
  ACCNO LE '9799'       Y   .       [  UNWANTED
A IGNORE                X   .       [  RECORDS
*SORT  AREA, ACCNO
*GO
```

A sample of the output appears below:

```
03/01/01     CUSTOMER BALANCE LISTING     PAGE    1

   AREA                  ACCOUNT        BALANCE
                         NUMBER

     21                     1144       £1511.20
     21                     1313        £151.50
     21                     8643        £109.22
     21                     9827        £210.80

     AREA 21 TOTAL                     £1982.72

     22                     3211         £16.84
     22                     3270        £128.62

     FINAL TOTAL                      £64303.77
```

Chapter 4
Fixed Logic and Printing

4.1 Field Specifying Characters (FSCs)

This chapter describes in detail:

- Field specifying characters
- FILETAB Fixed Logic
- FILETAB's internal storage areas
- Decision Points
- Tables accessed by the Fixed Logic
- Directives that control printing
- Printer control characters (Format Effectors).

As described in *Chapter 2*, a Field Specifying Character, usually abbreviated to FSC, is a single character taken from the character set available on the computer. It specifies whether a field is to be a control, totalling or transfer field. The default allocation of FSCs is as follows:

`M N O P Q R`	control fields
`0 1 2 3 4 5 6 7 8 9` `S T U V W X Y Z`	totalling fields
the remainder of the character set, that is, `A B C D E F` etc. except for `space '[ * ,`	transfer fields

Some of the non-alphanumeric characters are subject to certain restrictions, which are noted on the following pages. This default setting should be adequate for most reports. It may, however, be changed by the use of the `*ALTER` directive.

4.1.1 *ALTER Directive

The `*ALTER` directive is used to alter the characteristics of FSCs. It must appear before the `*INLIST` directive in the source statements.

The command has the following formats:

```
*ALTER FSCT
character string

*ALTER FSCT1
character string

*ALTER FSCT2
character string
```

`*ALTER FSCT` specifies a totalling field of the default size *(*see *Program Run-time *OPTIONs on page 3 of Chapter 5)*. `*ALTER FSCT1` specifies a totalling field of a single word and `*ALTER FSCT2` specifies a totalling field of two words. A two word field holds bigger numbers but requires more machine resources and is therefore less efficient.

Note: `'character string'` lists the characters to become totalling FSCs; it must begin in the first character position, and is terminated by the first space character, for example:

```
*ALTER FSCT
0123456789
```

sets FSCs 0-9 to totalling fields. FSCs S-Z become transfer fields.

All the characters available are listed below, each with their corresponding default FILETAB field, and the type or types of field to which each may be altered.

FSC	ASCII Values		Field Type		Note
	Dec	Hex	Default	Possible	
space	32	20	unallocated		*2*
!	33	21	Transfer	Total	*3*
.”	34	22	Transfer	Total	
$	36	24	Transfer	Total	*1*
%	37	25	Transfer	Total	
&	38	26	Transfer	Total	
(	40	28	Transfer	Total	
)	41	29	Transfer	Total	
*	42	2A	unallocated		*2*
+	43	2B	Transfer	Total	*1*
,	44	2C	unallocated		*2*
- (minus)	45	2D	Transfer	Total	*1*
.	46	2E	Transfer	Total	*1*
/	47	2F	Transfer	Total	*1*
0 to 9	48 to 57	30 to 39	Total	Transfer	
:	58	3A	Transfer	Total	
;	59	3B	Transfer	Total	
<	60	3C	Transfer	Total	
=	61	3D	Transfer	Total	
>	62	3E	Transfer	Total	
?	63	3F	Transfer	Total	*4*
@	64	40	Transfer	Total	
A to L	65 to 76	41 to 4C	Transfer	Total	
M to R	77 to 82	4D to 52	Control		
S to Z	83 to 90	53 to 5A	Total	Transfer	
[	91	5B	unallocated		*2*
\	92	5C	Transfer	Total	
]	93	5D	Transfer	Total	
^	94	5E	Transfer	Total	
_ (underline)	95	5F	Transfer	Total	
‘	96	60	unallocated		*2*
£	163	A3	Transfer	Total	*1*

Notes:

1 *When this character is used as an FSC, it overrides its normal editing function (see Chapter 4.7.14).*

2 *Unallocated characters cannot be used as FSCs.*

3 *Exclamation mark (!) is treated as a visible space in decision tables.*

4 *Question mark cannot be used as an FSC in decision tables because of its use in extended entry syntax.*

4.1.2 Dummy Totalling Fields

Each FSC used in a program is generally associated in the *INLIST with either a fieldname or a Field Definition. For example:

```
*INLIST
S  VALUE
M  PRODCODE
N  4/4
A  60:4
```

Here, the FSCs S and M have been associated with the *DICTIONARY fieldnames 'VALUE' and 'PRODCODE', and the FSCs N and A have been associated directly with fields in the Record Area. It is also possible, however, to enter an FSC without an associated field.

An FSC with no associated fieldname or Field Definition defines a 'dummy totalling field'. The FSC used must be one of the totalling FSCs. FILETAB allocates space for the accumulation of values set in this dummy totalling field by actions performed at the BREAK Decision Point (see *Decision Points on page 6*).

The following example shows the use of a dummy totalling field to calculate a percentage:

```
*INLIST
1 VALUE
2 DISCOUNT
3
M ACCNO
*DETAB BREAK M
A  *3  MV  *2      [ CALCULATES DISCOUNT AS A
   *3   *  100     [ PERCENTAGE OF VALUE
   *3   /  *1     [ FOR EACH ACCOUNT
```

Note: As the dummy totalling field has no value associated with it at the RECORD Decision Point, some value must be set in the dummy field before it is printed.

Dummy totalling fields are most advantageous in programs that include a *SORT. The records to be sorted are formed from *INLIST entries, but dummy totalling fields are not included. Consequently, the use of dummy totalling fields minimises the size of each sort record, allowing the *SORT to perform more efficiently.

4.1.3 Chaining Fields in the *INLIST

A control or transfer FSC may define more than one field in the *INLIST. Fields defined with the same FSC are chained together in the sequence in which they appear in the *INLIST. This facility is available only with character fields.

For example, if it is required to print Transaction Type, Reference Number, Invoice Number and Account Number, the fields could be defined in the *INLIST as:

```
*INLIST
A TT
B REFNO
C INVNO
D ACCNO
```

Alternatively, they could be defined in one of the three chaining formats:

```
*INLIST
A TT
A REFNO
A INVNO
A ACCNO

*INLIST
A TT REFNO INVNO ACCNO

*INLIST
A TT,A REFNO,A INVNO,A ACCNO
```

This facility is useful when a large number of fields are to be printed and there is a possibility that all the transfer FSCs are used. It is also useful for linking character control fields. For example, to use the Date, held in DDMMYY form in 20/6, as a control field in YYMMDD form:

```
*INLIST
M DATE+4/2
M DATE+2/2
M DATE/2
```

4.2 Decision Points

4.2.1 Introduction

The points at which FILETAB's Fixed Logic may be interrupted (in order to specify data selection and processing) are called Decision Points. So that they are entered at the appropriate points, defined decision tables are named corresponding to the names of the Decision Points, as follows:

`*DETAB START`	entered at the start of the run before the Main File is opened (`START` Decision Point)
`*DETAB RECORD`	entered each time a record has been input, normally from the Main File (`RECORD` Decision Point)
`*DETAB BREAK x`	entered after a control break test (`BREAK` Decision Point)
`*DETAB BREAKH x`	entered immediately prior to a control break heading (`BREAK` Decision Point).
`*DETAB ENDFILE`	entered at the end of the Main File (`ENDFILE` Decision Point)
`*DETAB PRINT`	entered each time a line is ready to be printed (`PRINT` Decision Point)

At these Decision Points, different areas of FILETAB storage may be referenced. These are known as the Record, Work, `*INLIST` and Print Areas. These areas, the Sort Slot and Decision Points, are described in the following pages.

4.2.2 Record Area

As each record is read automatically by FILETAB from the Main File, it is placed in an area known as the Record Area (*see also Chapter 2.5.1*). The first byte of the Record Area is byte zero. The start-positions of fields are defined by their byte displacement from byte zero. Fields in the Record Area may be referenced from decision tables by means of their Field Definitions or `*DICTIONARY` names. For example:

```
        RECTYPE     =  '05'
or      0/2         =  '05'

        CUSTYPE    LT  '07'
or      330/2      LT  '07'
```

The `*DICTIONARY` for this example would include entries like the following:

```
*DICTIONARY
 RECTYPE = 0/2
 CUSTYPE = 330/2
```

The Record Area may be used for processing data, for temporary storage, to contain record buffers for output files and additional input files (*see also Chapter 2.5.1*).

The Record Area is 10240 bytes long, the address of the last byte being 10239/1. The first byte of the Record Area, byte zero, is on a word boundary (*see Chapter 9.4*).

4.2.3 Work Area

Like the Record Area, the Work Area may be used for processing data, for temporary storage and to contain record buffers for output files and additional input files. Work Area contents are entirely under your control, through actions performed in decision tables.

Fields in the Work Area are defined 'backwards' relative to the last byte, which is -1. Fields may be referenced from decision tables by means of their Field Definitions or `*DICTIONARY` names. For example:

```
         WK1      MV   OUTSBAL
or       -8:8     MV   OUTSBAL

         WKDATE   S
or       -103/6   S
```

The `*DICTIONARY` for this example would include entries like the following:

```
*DICTIONARY
OUTSBAL  =  324:4
WK1      = -8:8
WKDATE   = -103/6
```

The Work Area is 10240 bytes long. *(See Chapter 8.3*) The address of the first byte of the Work Area is -10240/1. This byte is on a word boundary (*see Chapter 9.4*).

4.2.4 *INLIST Area

The *INLIST Area is an area of storage created by FILETAB from the fields defined by the *INLIST directive. FILETAB maintains the *INLIST Area by moving into it the appropriate fields from the Record and Work Areas. Fields in the *INLIST Area are referenced from decision tables entered at the BREAK Decision Point by means of their Field Specifying Character preceded by *.

```
*INLIST
A WKSWITCH
1 VALUE
M CODE
*DETAB BREAK M
C  *M  =  '100'    Y   Y   ELSE
   *A  =  '0'      Y   N   .
A  *1  MV 1000     X   .   .
   *1  MV 250      .   X   .
```

The *INLIST Area contains the values of the control fields; the values of the transfer fields; and the accumulated value of totalling fields for each control level defined in the *INLIST plus the List and Final (L and F) levels.

For example, if the *INLIST is defined as follows:

```
*INLIST
M ACCNO
N AREA
1 OUTSBAL
```

then four accumulation areas are created for the outstanding balance totals:

```
*1 List level
*1 M level
*1 N level
*1 Final level
```

Whenever a totalling field is accessed FILETAB ensures that it contains the appropriate value for the Current Control Level.

Totalling fields are held as binary in the *INLIST Area, irrespective of their original definition (see *ALTER FSCT in **ALTER Directive on page 2*).

When a transfer field is accessed at any break level higher than L, it contains the appropriate value from the last record processed.

4.2.5 Sort Slot

The Sort Slot is an area of storage used by FILETAB and is inaccessible to the user. This area is used by the Fixed Logic, to present records to, and receive records from, the *SORT.

4.2.6 Decision Table Entry

In order to interrupt the Fixed Logic at the appropriate Decision Point and to enter defined decision tables, the name of the relevant Decision Point is specified on the *DETAB directive. For example:

```
*DETAB RECORD
*DETAB BREAK L,M
```

Note: At the BREAK Decision Point, the control level(s) at which the table is to be entered must be specified as an additional operand. A decision table entered at a Decision Point may link to other decision tables by GOTO, CALL and IF verbs.

4.2.7 Returning to the Fixed Logic

When exit is made from a decision table, other than by GOTO, and there is no outstanding CALL or IF, control is returned to the FILETAB Fixed Logic; a CALL or IF is 'outstanding' when return has not yet been made from the called decision table.

```
*DETAB RECORD
C RECTYPE =   '05'      Y   Y   N
  STNAME  =   ' '       Y   N   .
A STNAME  MV  NAME      X   .   .
  STADD   MV  ADD       X   .   .
  GOTO REC99            .   .   X
*DETAB REC99
C ACCTOT  =  FILETOT        Y   N
A CALL     MATCH            X   .
  CALL     ERROR            .   X
```

*DETAB RECORD is entered automatically each time a record is read by FILETAB from the Main File.

Control is returned to FILETAB at the end of rules 1 and 2 in *DETAB RECORD; otherwise it is transferred to REC99. At the end of *DETAB REC99, control is returned to FILETAB.

The exit from MATCH or ERROR would cause control to return to *DETAB REC99, since they were CALLed decision tables. Control would then return to the Fixed Logic from *DETAB REC99, as in the preceding paragraph.

4.2.8 START Decision Point

The START Decision Point occurs at the start of processing, before the Main File is opened. The START decision table is optional and may be supplied to be entered by the Fixed Logic at this point. The table may enter other decision tables by means of CALL, IF or GOTO.

Control eventually returns to the Fixed Logic when exit is made from a decision table, other than by GOTO, and there is no outstanding CALL or IF; a CALL or IF is 'outstanding' when return has not yet been made from a called decision table.

When return to the Fixed Logic from the START Decision Point is made by EXIT F or IGNORE (explicit or implicit), processing immediately terminates; otherwise processing continues.

At the START Decision Point, constants in the Record and Work Areas may be initialised, and subsidiary files and fields in the Record and Work Areas may be accessed. The following is an example of a START decision table.

```
*DETAB START
C DD  EQ '01'                  Y Y N     [ DAY 01
  MM  EQ '01'                  Y N .     [ JANUARY
A RUNTYPE  MV 'YEAR END'       X . .     [ YEAR END RUN
  RUNTYPE  MV 'MONTH END'      . X .     [ MONTH END RUN
  RUNTYPE  MV 'WEEKLY'         . . X     [ WEEKLY RUN
```

This decision table determines the type of run to be performed from information stored in the current date. DD and MM are Reserved Words (*see* Chapter 7.1) which hold the current day and month numbers as two byte character fields. As the decision table does not contain a GOTO statement, control is returned to the Fixed Logic. If a program does not contain a *FILE directive, FILETAB assumes that the programmer is doing all necessary processing at *DETAB START. FILETAB must be prevented from entering its automatic reading routine, in one of four ways:

(1) By returning to the job control stream via the STOP verb.

(2) By EXITing FALSE from the START Decision Point.

(3) By moving 0 or -1 to RECLEN (*see below*), thereby implying that the end of the Main File has been reached.

(4) By moving a value less than -1 to RECLEN (*see below*), indicating that reading from the Main File is not required. The value of RECLEN is increased by 1, by the Fixed Logic, immediately before each call of *DETAB RECORD. The value of RECLEN must not be allowed to increase to -1, since a read from the Main File would be attempted. When no further entries to *DETAB RECORD are required, RECLEN must be set to zero (as in alternative 3 above).

4.2.9 RECORD Decision Point

The RECORD Decision Point occurs each time FILETAB has been presented with a record from the Main File, and at additional times, if Reserved Word RECLEN contains a value less than -1. A decision table called RECORD, may be optionally supplied, to be entered by the Fixed Logic whenever a RECORD Decision Point occurs. The RECORD decision table may enter other decision tables by means of CALL, IF or GOTO. Control eventually returns to the Fixed Logic when exit is made from a decision table, other than by GOTO, and there is no outstanding CALL or

IF; a CALL or IF is 'outstanding' when return has not yet been made from the called decision table.

If return to the Fixed Logic from a RECORD Decision Point has been made other than by EXIT F or IGNORE (explicit or implicit), and any of the following conditions apply, the record currently in the Record Area is passed forward for tabulation or sorting:

- The value of Reserved Word RECLEN is left unchanged at the RECORD Decision Point
- The value of Reserved Word RECLEN is changed to a new value less than -1
- The value of Reserved Word RECLEN is positive, both immediately before entry to the RECORD decision table and immediately after return to the Fixed Logic

In all other cases, the record currently in the Record Area is not passed forward for tabulation or sorting. Instead, the ENDFILE Decision Point occurs. The following is an example of a RECORD decision table.

```
*DETAB  RECORD
C  RECTYPE  =  '?'    01   02
A  IGNORE             .     X
```

If RECTYPE is 01, the record is passed forward for printing and totalling. If RECTYPE is 02, the record is explicitly ignored and is not passed forward for printing and totalling.

If RECTYPE is neither 01 nor 02, the record is ignored by the implied ELSE/IGNORE action and is not passed forward for printing and totalling.

At the RECORD Decision Point, the Record and Work Areas may be referenced.

The Fixed Logic can be prevented from reading a record from the Main File before entry to the RECORD Decision Point. This is done by setting the Reserved Word RECLEN to a value smaller (that is, more negative) than -1 at either the START Decision Point or at a previous RECORD Decision Point (*see below*).

4.2.10 Use of RECLEN at the RECORD Decision Point

If RECLEN is set to -n (where n is greater than 1), at the START or RECORD Decision Points, reading of records from the Main File is temporarily suspended, but the rest of the Fixed Logic is obeyed normally.

Each time FILETAB is ready to receive another input record, a READ from the Main File is attempted, unless Reserved Word RECLEN has a value less than -1. In that case, the value of RECLEN is increased by 1, and no READ from the Main File is attempted. Thus, if RECLEN is set to -n when a record is first processed at the RECORD Decision Point, the same record is presented a further n-1 times. Changes made to data in a record remain effective when it is re-presented.

A RECORD Decision Point then occurs.

In the following example, a file contains data in this format:

```
Part no.  Supplier code 1  Supplier code 2  Supplier code 3
```

A listing is required showing the parts available from each supplier. Using RECLEN, this can be done as follows:

```
*DICTIONARY
 PART  =  4:4
 SUP1  =  8:4
 SUP2  = 12:4
 SUP3  = 16:4
 SUPWK = -4:4
*
*INLIST
A  PART
M  SUPWK
*
*HEAD M
 'MMMMMMM'
*OUT L
   AAAAAA
*DETAB RECORD
C  RECLEN  >  0      Y  N  N
   RECLEN  = -2      .  Y  N
A  RECLEN MV -3      X  .  .
   SUPWK  MV SUP?    1  2  3
*
*SORT
*STOP
```

Rule 1 of *DETAB RECORD is obeyed when RECLEN is greater than 0, that is, when a record has just been read from the Main File. The actions for rule 1 set RECLEN to -3 (to instruct the Fixed Logic to present the record twice more) and move the first supplier code into the field associated with the FSC M.

Rule 2 is obeyed when RECLEN is equal to -2. This occurs when the record is presented for the second time, since RECLEN will have been incremented from -3 to -2, by the Fixed Logic, immediately before *DETAB RECORD was entered. The second supplier code is moved into the field associated with the FSC M.

On the third entry to *DETAB RECORD, RECLEN equals -1, and the third rule is obeyed. Consequently, supplier code 3 is processed.

After processing supplier code 3, the Fixed Logic changes the value of RECLEN to 1 (that is, to a positive value). Because RECLEN is positive, the Fixed Logic reads the next record from the Main File before the next entry to *DETAB RECORD. The process is thus repeated.

At End-of-Main-File, RECLEN is set to -2 by FILETAB's Fixed Logic, before entry to the ENDFILE Decision Point (*see below*). If no EXIT F or IGNORE is made from the ENDFILE Decision Point, and RECLEN is still less than -1, RECLEN is incremented by 1 and a further RECORD Decision Point occurs.

Further RECORD Decision Points occur, until the value of RECLEN is no longer less than -1. This facility allows further records, from sources other than the Main File, to be presented for tabulation.

4.2.11 BREAKH Decision Point

If a *DETAB BREAKH decision table is specified at a particular control level, FILETAB enters it when a control break occurs at that level. The decision table is entered at the beginning of the control group and the user may refer to the *INLIST and work Areas.

```
*INLIST
M DIVNO, A DIVNAME
*DETAB BREAKH M
C *M EQ '?'          1        2        3       4       ELSE
A *A MV '?'        NORTH    SOUTH    EAST    WEST    INVALID
*HEAD M CH1,2,55
                     SALES FOR 'AAAAAAA' DIVISION
```

The *INLIST area contains the control, transfer and totalling fields from the first record in the control group. The BREAKH Decision Point allows users to process fields before they are printed according to the *HEAD detail statements, in particular by using fields in the *INLIST area as a key to LOOKUP entries in a *TABLE (see *Tables on page 24 of Chapter 6*). The description obtained can then be included in any headings that are to be printed.

After a *DETAB BREAKH decision table has been processed, control is returned to FILETAB:

(1) In a decision table entered other than by CALL or IF, a rule is satisfied that does not contain a GOTO action. The corresponding *HEAD is then processed.

(2) In a decision table entered other than by CALL or IF, a rule is satisfied that contains an IGNORE action. the corresponding *HEAD is not printed. the IGNORE action may be explicit or implicit (*see Incomplete Decision Tables on page 12 of Chapter 3*).

4.2.12 BREAK Decision Point

The *DETAB BREAK directive, which is associated with this Decision Point, defines decision tables to be entered when specified control breaks occur. Consequently, fields can be processed before they are printed according to the *OUT detail statements, or printing can be suppressed altogether. The format is:

```
*DETAB BREAK  control level(s)
```

'control level(s)' specifies the control level(s) at which this decision table is to be entered. When more than one control level is specified, they should be separated by commas with no spaces.

Fields in the *INLIST can be accessed within this decision table (or within others called by it) by means of *FSCs.

The decision table is entered if a control break occurs at any level specified on the directive, or at any higher level. When more than one level is specified, the decision table is entered once for each specified level (but not for levels higher than the current control break). More than one *DETAB BREAK directive may be specified in a program; however, no control level character may appear on more than one *DETAB BREAK directive. The appropriate *DETAB BREAK decision tables are entered (where defined) in ascending order of control level (that is, from L level to the current break level). The following example illustrates the use of this directive:

```
*INLIST
M ACCNO
1 OUTSBAL
*DETAB BREAK  M
C *1  GE  100    Y  N    [ IF BALANCE EXCEEDS £100,
A *1  *   92     X  .    [  DISCOUNT BY 8%
  *1  /   100    X  .
```

The value in *1 in the above decision table is the accumulated total of the outstanding balance for records since the last break of M or any higher level.

Care should be taken with coding at a BREAK Decision Point that accesses Record or Work Area fields set up at the RECORD Decision Point, since their contents may already have been overwritten by fields associated with the next record.

Two Reserved Words (*see Chapter 7.1*), BREAK and LEVEL, are of particular significance at the BREAK Decision Point.

BREAK contains the level of the current control break. If, for example, the control levels M, N and O are specified, and there is a change of major control key (BREAK O), BREAK contains the character 'O'. This may be tested at *DETAB BREAK at any level (for example, in *DETAB BREAK M). The level of the current control break can thus be determined.

LEVEL contains the current control level being processed, that is, either a letter between L and the current control break, or F. LEVEL is useful in determining the current control level, when multiple control levels are specified on the *DETAB BREAK directive, for example:

```
*DETAB BREAK M,N,O,P
```

Control is returned to the Fixed Logic when exit is made from a decision table, other than by GOTO, and there is no outstanding CALL or IF; a CALL or IF is 'outstanding' when return has not yet been made from a called decision table. On return to the Fixed Logic, the following action is taken:

Provided that control has not been returned by an EXIT F or IGNORE, any *OUT lines required for the current control level is printed.

If control has been returned by EXIT F or IGNORE, *OUT lines for the current control level are not printed. For example:

```
*DETAB   BREAK    M
C  *1   GT  1000  Y   N
A  IGNORE         .   X
```

In the above table, if the value in *1 is less than or equal to 1000 the *OUT M line(s) are not printed.

Note: The appropriate totals are still accumulated.

Dummy totalling fields (*see Chapter 4.1.2*) may be referenced at the BREAK Decision Point by use of the appropriate totalling FSC preceded by *.
An IGNORE at the BREAK level does not affect the totalling:

```
*INLIST
 M  DEPT,1  AMOUNT
*OUT L
   1111111
*OUT M
MM  1111111
*DETAB BREAK L
C  *M  EQ  '05'    Y  N
A  IGNORE          X  .
```

Even though, in the above example, the detailed records for department 05 have not been printed, the total for the department, printed at BREAK M, is still valid.

4.2.13 ENDFILE Decision Point

The ENDFILE Decision Point occurs either when the end of the Main File is encountered or when the value held in Reserved Word RECLEN is changed to zero or -1 at either the START or RECORD Decision Point. The ENDFILE decision table is optional and may be supplied, to be entered by the Fixed Logic when the ENDFILE Decision Point is reached.

Before the ENDFILE decision table is entered, the value -2 is moved to RECLEN by the Fixed Logic. The ENDFILE decision table may enter other decision tables by means of CALL, IF or GOTO. Control eventually returns to the Fixed Logic when exit is made from a decision table, other than by GOTO, and there is no outstanding CALL or IF; a CALL or IF is ’outstanding’ when return has not yet been made from the called decision table.

On return to the Fixed Logic from the ENDFILE Decision Point the following action is taken:

If return was made other than by EXIT F or IGNORE (explicit or implicit), the value of RECLEN is checked. If this value is less than -1, it is increased by 1 and a further RECORD Decision Point occurs. A further record is presented for tabulation or sorting if return is made from this RECORD Decision Point other than by EXIT F or IGNORE (implicit or

explicit). These three steps are repeated until RECLEN is found to be not less than -1, in which case no further RECORD Decision Point occurs.

If return was made by EXIT F or IGNORE (explicit or implicit) no subsequent RECORD Decision Point occur.

The Record and Work Areas may only be referenced at the ENDFILE Decision Point.

4.2.14 Closing The Main File Early

The Main File can be closed before end-of-file has been reached, by changing RECLEN to zero or -1, before exit from the RECORD Decision Point. This prevents FILETAB reading further records from the Main File and effectively simulates End-of-Main-File by causing an ENDFILE Decision Point to occur immediately after returning from the RECORD Decision Point (*see Example below*). Closure of the Main File does not occur if RECLEN holds -1 both on entry to, and on exit from, the RECORD Decision Point. It is recommended that RECLEN be set to zero, rather than -1, to avoid confusion with the case where RECLEN is increased from -2 to -1 by the Fixed Logic, before *DETAB RECORD is called (*see 'Use of RECLEN at the RECORD Decision Point' above*). The following example indicates the use of RECLEN to close the Main File early.

```
*DICTIONARY
 AREA     = 16/16
 SALESMAN =  0/16
 SALES    = 32:4
 VALUE    = 36:4
*
*INLIST
 M AREA,A SALESMAN,1 SALES,2 VALUE
*
*DETAB RECORD
C  AREA/10  =  'NORTH WEST'      Y  N
A  RECLEN   Z                    X  .
*
*HEAD M CH0,2
AREA 'MMMMMMMMMMMMMMMM'
*OUT L
  AAAAAAAAAAAAAAAA   1,111,111   2,222,222.22
*STOP
```

This would print the required details up to, but not including, those for 'NORTH WEST'. When 'NORTH WEST' was reached, RECLEN would be zeroised, closing the Main File.

4.2.15 PRINT Decision Point

If *DETAB PRINT is specified, it is entered each time FILETAB is ready to print a line of output. Any specified actions, including transfer of control to other decision tables, are obeyed. When exit is made from a decision table, other than by GOTO, and there is no outstanding CALL or IF, control is returned to the FILETAB Fixed Logic; a CALL or IF is 'outstanding' when return has not yet been made from a called decision table. On return to the Fixed Logic, the following action is taken:

Provided that control has not been returned by an EXIT F or IGNORE, the current line in the Print Area is prefixed with the appropriate Format Effector (*see Chapter 4.8*), and is output to the line printer spool file; the count of the number of lines still available for printing on the current page (*see Reserved Word LINES, Chapter 7.1*) is reduced. Trailing spaces are automatically suppressed.

If control has been returned by EXIT F or IGNORE, the output line is not printed. The count of the lines still available for printing on the current page (LINES) remains unchanged.

```
*DETAB PRINT
C PP14-21  EQ  'DEPT. 99'  Y  N    [ SUPPRESS PRINTING OF
A IGNORE                   X  .    [  DEPARTMENT 99
```

At the PRINT Decision Point, all areas are accessible. However, it is normal to refer only to the Print Area (PP1 to PP160) and the Work Area. Any data referenced in the Work Area should be treated as being in local working storage. Data set up in the Work Area at the RECORD Decision Point should not be accessed at the PRINT Decision Point, since the data refers to different records.

At *DETAB PRINT, four bytes are available that precede the print buffer. The contents of these bytes are as follows:

PP-3 /1	Holds the control level of the line being printed.
PP-2 /1	Holds 'H' if the line being printed is a heading, 'O' if it is an output line, or 'T' if it is a title.
PP-1 :1	Holds the number of the line within the print group.
PP0 :1	Holds a FILETAB printer control character.

4.3 Tables Accessed by the Fixed Logic

In FILETAB, tables may be created within sets of source statements, to associate 'keys' with 'descriptors'. Any number of these tables may be created within a set of source statements, and each table may contain any number of entries. A character field may then be compared with such a table, to test for the presence of a particular key, or to access the descriptor associated with the key, or to do both.

For example, products may be held in coded form on a particular file. A table can be produced which has a list of product codes as the keys. A meaningful description of each product can be associated with each key; this is known as a descriptor. This would allow for a description of the product to be printed, whenever an entry in the table corresponded to a product code appearing in the record.

Two directives are used to define tables: *LOOKUP and *TABLE. When *LOOKUP is used, FILETAB accesses the table at the required Decision Point, automatically. When *TABLE is used, the point at which access is

made is entirely under the user's control. *TABLE *is described in Chapter Chapter 6.*

4.3.1 *LOOKUP Directives

When a RECORD, BREAK or PRINT decision point is reached FILETAB looks for a corresponding *LOOKUP table and retrieves a descriptor, pointed to by a key in the *LOOKUP table definition.

4.3.1.1 *LOOKUP RECORD

```
*LOOKUP RECORD   keyfield,descriptorfield
entry format statement
text statements
```

'keyfield' specifies the area to contain the key. It must be a *DICTIONARY name or a character field definition.

'descriptorfield' specifies the area into which the descriptor is read. It must be a *DICTIONARY name defining a character field, or an explicit character field definition.

'entry format statement' defines the position of keys and descriptors on the following 'text statements', by means of Ks and Ds. Each K represents one character of key, and each D represents one character of descriptor. The Ks must always precede the Ds. The total number of Ks and Ds must not exceed eighty characters. Spaces must not be embedded within the group of Ks or the group of Ds, but may appear between the two groups.

For example:

Valid	Invalid
KKKDDD KKK DDD KKK	K KKDDD KKKD DD DDDKKK DDD

text statements follow the entry format statement, and specify the contents of the table entries. The position of the text must correspond to that specified on the entry format statement. If, when the table is being compiled, the keys are not presented in sequence, the entries are sorted into ascending order. Duplicate keys are accepted as valid entries, but the sequence of such keys is not determinate. Any number of text statements may be specified, the last of which should have a dummy key value. When the *LOOKUP is performed, if the specified key cannot be found, the descriptor associated with the dummy value is returned. The dummy key value should be outside the normal range of key values expected.

The following example illustrates the use of *LOOKUP RECORD:

```
*FILE CUSTFILE RL=400  NAME=custfile.txt
*DICTIONARY
```

```
CUSTCODE = 16/4
CUSTNAME = 500/12
*INLIST
.
.
*DETAB RECORD
.
.
*LOOKUP RECORD CUSTCODE,CUSTNAME
KKKK DDDDDDDDDDDD
1234 XYZ.CO.LTD
1338 ACME CO
2236 BETA LTD
.
.
NONE NOT IN TABLE
*GO
```

Here, each time a record is presented from the Main File, and before *DETAB RECORD, or any other decision table at that level is entered, the customer code held in 16/4 is used as a key to search the table. If the key is present, the associated descriptor is placed in the customer name field (500/12); if, however, the key is absent, the legend 'NOT IN TABLE' is placed there. Thus, if a record is read in which 16/4 is set to '1338', CUSTNAME will contain ACME CO after the *LOOKUP RECORD has been processed.

*Note: *LOOKUP RECORD is always performed before *DETAB RECORD is entered.*

4.3.1.2 *LOOKUP BREAK x

```
*LOOKUP BREAK x   keyfield,descriptorfield
entry format statement
text statements
```

x indicates the appropriate control level. Only one level may be specified in any one *LOOKUP BREAK directive, but *LOOKUP BREAK directive is allowed for each break level.

'keyfield' specifies the area which contains the key. It must be an FSC preceded by an asterisk.

*Note: The *LOOKUP table (including, all *LOOKUP BREAK directives) must be defined before any *HEAD or *OUT commands which use the FSC. The *OUT is only flushed out if the *LOOKUP appears before the *OUT. If a *LOOKUP appears after a *OUT a warning message is generated at compilation time.*

'descriptorfield' specifies the area into which the descriptor is read. It is normally a character field definition or an *FSC. In the case of *FSC, because FILETAB generates an *INLIST entry, the FSC must not have been defined previously.

*Note: As with *LOOKUP RECORD, the *LOOKUP BREAK x is performed before any BREAK decision table at that level is entered. All other details are as for *LOOKUP RECORD.*

4.3.1.3 *LOOKUP PRINT

```
*LOOKUP PRINT   keyfield,descriptorfield
entry format statement
text statements
```

'keyfield' must be a *DICTIONARY name or a character field definition, defining an area in the print buffer which holds the key.

'descriptorfield' specifies the area into which the descriptor is read. It would normally be a *DICTIONARY name or a character field definition, and would normally refer to an area within the print line.

*Note: *LOOKUP PRINT is always performed before *DETAB PRINT is entered.*

All other details are as for *LOOKUP RECORD.

4.4 Structure of the Fixed Logic

4.4.1 Introduction

The following pages list the steps through which the FILETAB Fixed Logic proceeds. The first cycle described is that undertaken when the program in question contains no *SORT directive. This is followed by the cycle as it is performed when a *SORT directive is present. The third cycle described applies when no *INLIST directive is present.

Structure of the Fixed Logic on page 20 also includes a description of the 'totals-only' *SORT facility.

The Decision Points mentioned in this section are those noted in *Decision Table and Fixed Tabulating Logic on page 33 of Chapter 3*. Full details of these Decision Points may be found in *Decision Points on page 6*.

4.4.2 Fixed Logic with *INLIST but no *SORT

4.4.2.1 START DECISION POINT

(1) Set RECOUNT, LINES and PAGE equal to zero set RECLEN and COUNT equal to one; set DD,MM,YY and HH,MM,SS to the date and time of the program run. Zeroise totalling field values for all levels in the *INLIST.

(2) Print *TITLE, and then reset PAGE equal to zero.

(3) Enter *DETAB START. If return to the Fixed Logic is made by IGNORE or EXIT F, terminate the run. If RECLEN equals -1 or zero, go to '**ENDFILE DECISION POINT**' (Step 28).

(4) Go to '**RECORD DECISION POINT**' (Step 14).

4.4.2.2 PRINT *HEADs

(5) Zeroise L level totalling field values in the *INLIST Area, and move record from the Record Area to the *INLIST Area. Totalling fields overwrite the L level values stored from the previous record (if any).

(6) If *OPTION RPNx is present, and the level of the Current Control Break is higher than or equal to x, reset the current page number (held in the Reserved Word PAGE) to zero.

(7) Set the Current Control Level equal to the level of the Current Control Break.

(8) If there is a *HEAD L CHn in the program, and there is insufficient room on the current page to print all the *HEADs for the level of the Current Control Break down to the M level, perform *LOOKUP BREAK L and print *HEAD L CHn, beginning at line n (n being greater than or equal to zero) on the next page.

(9) If there is no *HEAD L CHn in the program, and there is insufficient room on the current page to print *HEADs for the level of the Current Control Break down to the L level, throw to the top of the next page.

(10) Perform *LOOKUP BREAK for the Current Control Level.

Enter *DETAB BREAKH. If EXIT FALSE go to step 12.

If *HEAD for the Current Control Level has already been printed at another Control Level during the current Fixed Logic cycle, go to Step 12.

(11) Print *HEAD for the Current Control Level.

(12) If the Current Control Level is L, go to '**RECORD DECISION POINT**' (Step 14).

(13) Set the Current Control Level equal to the next lower level, and repeat from Step 9.

4.4.2.3 RECORD DECISION POINT

(14) If RECLEN is greater than zero, go to Step 16.

(15) If RECLEN is less than -1, go to Step 22.

(16) Read a record from the Main File (that is, *FILE) into the Record Area, starting from address zero.

If the read is successful, set RECLEN equal to the size in bytes of the record read, add one to RECOUNT, and go to Step 17.

If the read is unsuccessful because the end of the Main File has been reached, go to '**ENDFILE DECISION POINT**' (Step 28).

If the read is unsuccessful for any other reason, output an error message and terminate the run.

(17) Perform *LOOKUP RECORD.

(18) Enter *DETAB RECORD.

(19) If RECLEN equals zero or -1, go to '**ENDFILE DECISION POINT**' (Step 28).

(20) If return from Step 18 to the Fixed Logic was made by IGNORE or EXIT F, repeat from Step 14.

(21) If no record has yet been moved to the *INLIST Area, set the level of the Current Control Break to 'F' and go to '**PRINT *HEADs**' (Step 5); otherwise go to '**CONTROL BREAK TESTING**' (Step 33).

(22) Add one to RECLEN; record the new value of RECLEN in an internal location.

(23) Perform *LOOKUP RECORD.

(24) Enter *DETAB RECORD.

(25) If RECLEN is less than -1, go to Step 20.

(26) If RECLEN equals -1, and the recorded value of RECLEN is also -1, go to Step 27; otherwise go to '**ENDFILE DECISION POINT**' (Step 28).

(27) If '**ENDFILE DECISION POINT**' has been reached previously, set RECLEN equal to zero; otherwise set RECLEN equal to one. Go to Step 20.

4.4.2.4 ENDFILE DECISION POINT

(28) If this point has been reached before, go to '**END OF INPUT RECORD SET**' (Step 32).

(29) Set RECLEN equal to -2.

(30) Enter *DETAB ENDFILE. (If no *DETAB ENDFILE is supplied in the program, the Compiler generates a *DETAB ENDFILE containing the single action IGNORE.)

(31) If return to the Fixed Logic is made by EXIT T, go to '**RECORD DECISION POINT**' (Step 14).

4.4.2.5 END OF INPUT RECORD SET

(32) If no record has yet been moved to the *INLIST Area from the Record Area, output the message 'NO DATA SELECTED', and end the run; otherwise set the level of the Current Control Break to 'F', set the Current Control Level to 'L', and go to '**BREAK DECISION POINT**' (Step 34).

4.4.2.6 CONTROL BREAK TESTING

(33) Determine the level of the Control Break by comparing the Control Fields of the record currently in the Record Area with the corresponding Control Fields of the record currently in the `*INLIST` Area, starting with the Control Fields for the highest Control Level.

If inequality is found in a Control Field pair, set the level of the Current Control Break equal to the Control Field Specifying Character for the highest Control Level at which inequality occurs. Store this Control character in `BREAK`.

If all Control field pairs are equal, set the level of the Current Control Break equal to `L`. Store the Control character `L` in `BREAK`.

Set the Current Control Level to `L`, and store `L` in `LEVEL`.

4.4.2.7 BREAK DECISION POINT

(34) Enter `*DETAB BREAK` for the Current Control Level, and, if return is made by `IGNORE` or `EXIT F`, go to '**ROLL TOTALS**' (Step 38).

4.4.2.8 PRINT *OUTs

(35) If there is a `*HEAD L CHn` in the program, and there is insufficient room on the current page to print `*OUT` for the Current Control Level, perform *LOOKUP BREAK L and print `*HEAD L CHn`, beginning at line n (n being greater than or equal to zero) on the next page.

If there is no `*HEAD L CHn`, and there is insufficient room on the current page to print `*OUT` for the Current Control Level, throw to the top of the next page.

(36) Print `*OUT` for the Current Control Level.

(37) If the Current Control Level is '`F`', terminate the run.

4.4.2.9 ROLL TOTALS

(38) Roll Totalling Fields in the `*INLIST` Area by adding the value held for each Totalling Field for the Current Control Level into the corresponding value for the next higher Control Level; then zeroise the value held for each Totalling Field for the Current Control Level.

(39) If the Current Control Level equals the level of the Current Control Break, go to '**PRINT *HEADs**' (Step 5).

(40) Raise the Current Control Level to the next higher level, store the Control character for the new level in LEVEL, and go to '**BREAK DECISION POINT**' (Step 34).

4.4.3 Note on Print Logic

When a *TITLE, *HEAD or *OUT print block is to be printed, the first line is moved into PP1-160, and *LOOKUP PRINT is performed. *DETAB PRINT is then entered. The line in PP1-160 is printed, prefixed by the appropriate printer control character (*see Chapter 4.8*), except where EXIT F or IGNORE has been obeyed. In either case, the printer control character in PP0 is reset to binary 1 (single-line throw). This process is repeated for each line in the print block.

4.4.4 Fixed Logic with *INLIST and *SORT

4.4.4.1 START DECISION POINT

(1) Set RECOUNT, LINES and PAGE equal to zero; set RECLEN and COUNT equal to one; set DD,MM,YY and HH,MM,SS to the date and time of the program run; zeroise totalling field values for all levels in the *INLIST.

(2) Print *TITLE, and then reset PAGE equal to zero.

(3) Enter *DETAB START. If return to the Fixed Logic is made by IGNORE or EXIT F, terminate the run. If RECLEN equals -1 or zero, go to '**ENDFILE DECISION POINT**' (Step 18).

4.4.4.2 RECORD DECISION POINT

(4) If RECLEN is greater than zero, go to Step 6.

(5) If RECLEN is less than -1, go to Step 12.

(6) Read a record from the Main File (that is, *FILE) into the Record Area, starting from address zero.

If the read is successful, set RECLEN equal to the size in bytes of the record read, add one to RECOUNT, and go to Step 7.

If the read is unsuccessful because the end of the Main File has been reached, go to '**ENDFILE DECISION POINT**' (Step 18).

If the read is unsuccessful for any other reason, output an error message and terminate the run.

(7) Perform *LOOKUP RECORD.

(8) Enter *DETAB RECORD.

(9) If RECLEN equals zero or -1, go to '**ENDFILE DECISION POINT**' (Step 18).

(10) If return from *DETAB RECORD to the Fixed Logic was made by IGNORE or EXIT F, repeat from Step 4.

4.4.4.3 INPUT TO THE SORT

(11) Input the record currently in the Record Area to the SORT. Repeat from Step 4.

(12) Add one to RECLEN; record the new value of RECLEN in an internal location.

(13) Perform *LOOKUP RECORD.

(14) Enter *DETAB RECORD.

(15) If RECLEN is less than -1, go to Step 10.

(16) If RECLEN equals -1, and the recorded value of RECLEN is also -1, go to Step 17; otherwise go to '**ENDFILE DECISION POINT**' (Step 18).

(17) If '**ENDFILE DECISION POINT**' has been reached previously, set RECLEN equal to zero; otherwise set RECLEN equal to one. Go to Step 10.

4.4.4.4 ENDFILE DECISION POINT

(18) If this point has been reached before, go to '**END OF INPUT RECORD SET**' (Step 22).

(19) Set RECLEN equal to -2.

(20) Enter *DETAB ENDFILE. (If no *DETAB ENDFILE is supplied in the program, the Compiler generates a *DETAB ENDFILE containing the single action 'IGNORE'.)

(21) If return to the Fixed Logic is made by EXIT T, go to '**RECORD DECISION POINT**' (Step 4).

4.4.4.5 END OF INPUT RECORD SET

(22) If no records have been input for sorting, output the message 'NO DATA SELECTED', and terminate the run.

4.4.4.6 SORT

(23) Sort the input records.

(24) Set the level of the Current Control Break to F.

(25) Obtain first record in sorted sequence in the Sort Slot.

4.4.4.7 PRINT *HEADs

(26) Zeroise L level dummy totalling field values in the *INLIST Area, and move record from the Sort Slot to the *INLIST Area. Totalling fields overwrite the L level values stored from the previous record (if any).

(27) If *OPTION RPNx is present, and the level of the Current Control Break is higher than or equal to x, reset the current page number (held in the Reserved Word PAGE) to zero.

(28) Set the Current Control Level equal to the level of the Current Control Break.

(29) If there is a *HEAD L CHn in the program, and there is insufficient room on the current page to print all the *HEADs for the level of the Current Control Break down to the M level, perform *LOOKUP BREAK L and print *HEAD L CHn, beginning at line n (n being greater than or equal to zero) on the next page.

If there is no *HEAD L CHn in the program, and there is insufficient room on the current page to print *HEADs for the level of the Current Control Break down to the L level, throw to the top of the next page.

(30) Perform *LOOKUP BREAK for the Current Control Level.

Enter *DETAB BREAKH. If EXIT FALSE go to step 33.

(31) If *HEAD for the Current Control Level has already been printed at another Control Level during the current Fixed Logic cycle, go to Step 33.

(32) Print *HEAD for the Current Control Level.

(33) If the Current Control Level is L, go to '**GET NEXT SORTED RECORD**' (Step 35).

(34) Set the Current Control Level equal to the next lower level, and repeat from Step 30.

4.4.4.8 GET NEXT SORTED RECORD

(35) Read next record in sorted sequence into the Sort Slot. If the read fails because there are no more records, set the level of the Current Control Break to F, set the Current Control Level to L and go to '**BREAK DECISION POINT**' (Step 37). If the read fails for any other reason, output an error message and terminate the run.

4.4.4.9 CONTROL BREAK TESTING

(36) Determine the level of the Control Break by comparing the Control Fields of the record currently in the Sort Slot with the corresponding Control Fields of the record currently in the *INLIST Area, starting with the Control Fields for the highest Control Level.

If inequality is found in a Control Field pair, set the level of the Current Control Break equal to the Control Field Specifying Character for the highest Control Level at which inequality occurs. Store this Control Character in BREAK.

If no inequality is found, set the level of the Current Control Break equal to L. Store the Control Character L in BREAK. Set the Current Control Level to L, and store L in LEVEL.

4.4.4.10 BREAK DECISION POINT

(37) Enter *DETAB BREAK for the Current Control Level, and, if return is made by IGNORE or EXIT F, go to '**ROLL TOTALS**' (Step 41).

4.4.4.11 PRINT *OUTs

(38) If there is a *HEAD L CHn in the program, and there is insufficient room on the current page to print *OUT for the Current Control Level, perform *LOOKUP BREAK L and print *HEAD L CHn, beginning at line n (n being greater than or equal to zero) on the next page.

If there is no *HEAD L CHn, and there is insufficient room on the current page to print *OUT for the Current Control Level, throw to the top of the next page.

(39) Print *OUT for the Current Control Level.

(40) If the Current Control Level is F, terminate the run.

4.4.4.12 ROLL TOTALS

(41) Roll Totalling Fields in the *INLIST Area by adding the value held for each Totalling Field for the Current Control Level into the corresponding value for the next higher Control Level; then zeroise the value held for each Totalling Field for the Current Control Level.

(42) If the Current Control Level equals the level of the Current Control Break, go to '**PRINT *HEADs**' (Step 26).

(43) Raise the Current Control Level to the next higher level, store the Control Character for that level in LEVEL and go to '**BREAK DECISION POINT**' (Step 37).

4.4.5 Note on Print Logic

When a *TITLE, *HEAD or *OUT print block is to be printed, the first line is moved into PP1-160, and *LOOKUP PRINT is performed. *DETAB PRINT is then entered. The line in PP1-160 is printed, prefixed by the appropriate printer control character (*see Chapter 4.8*), except where EXIT F or IGNORE has been obeyed. In either case, the printer control character in PP0 is reset to binary 1 (single-line throw). This process is repeated for each line in the print block.

4.4.6 Diagram of Sort Record Layout for 'totals-only' *SORT

k	nn	g	tot1...totn

k represents the sort key.

nn is the area containing fields not to be totalled by the sort.

g represents the filler gap to the next four-byte address boundary.

tot1 to totn are eight-byte binary totalling fields.

A totals-only *SORT is performed in-store whenever possible. (If it is not possible to do this, there may be more than one sort record with the same

key among the output records, but, since there is no printing at the L level, such records are amalgamated by the Fixed Logic before printing.) The following example would produce a totals-only *SORT.

```
*PROGRAM TEST3
*OPTION PE,NSDR
*FILE TRANSDY RL=64  NAME=transdy.txt
*DICTIONARY
 TACCNO   = 4/4             [ TRANSDY ACCOUNT NUMBER
 TVALUE   = 28:4            [ TRANSACTION VALUE
*INLIST
 M TACCNO
 2 TVALUE
*HEAD L CH1,3,57
    DD/MM/YY                ANALYSIS OF ORDER VALUES

    ACCOUNT NO.             TOTAL VALUE
*OUT M 0,2
      MMMM                  £22222222.22
*OUT F 1,0
                           '------------'

    'TOTAL VALUES'         £2222222222.22
*SORT
*STOP
```

4.4.7 Fixed Logic with No *INLIST

4.4.7.1 START DECISION POINT

(1) Set RECOUNT, LINES and PAGE equal to zero; set RECLEN and COUNT equal to zero; set DD,MM,YY and HH,MM,SS to the date and time of the program run.

(2) Print *TITLE, and then reset PAGE to zero.

(3) Enter *DETAB START. If return to the Fixed Logic is made by IGNORE or EXIT F, terminate the run. If RECLEN equals -1 or zero, go to '**ENDFILE DECISION POINT**' (Step 16).

4.4.7.2 RECORD DECISION POINT

(4) If RECLEN is greater than zero, go to Step 6.

(5) If RECLEN is less than -1, go to Step 10.

(6) Read a record from the Main File (that is, *FILE) into the Record Area, starting from address zero.

If the read is successful, set RECLEN equal to the size in bytes of the record read, add one to RECOUNT, and go to Step 7.

If the read is unsuccessful because the end of the Main File has been reached, go to '**ENDFILE DECISION POINT**' (Step 16).

If the read is unsuccessful for any other reason, output an error message and terminate the run.

(7) Perform *LOOKUP RECORD.

(8) Enter *DETAB RECORD.

(9) If RECLEN equals zero or -1, go to '**ENDFILE DECISION POINT**' (Step 16); otherwise repeat from Step 4.

(10) Add one to RECLEN, and record the new value of RECLEN in an internal location.

(11) Perform *LOOKUP RECORD.

(12) Enter *DETAB RECORD.

(13) If RECLEN is less than -1, go to Step 4.

(14) If RECLEN equals -1, and the recorded value of RECLEN is also -1, go to Step 15; otherwise go to '**ENDFILE DECISION POINT**' (Step 16).

(15) If **ENDFILE DECISION POINT** has been reached previously, set RECLEN equal to zero; otherwise set RECLEN equal to one. Go to Step 4.

4.4.7.3 ENDFILE DECISION POINT

(16) If this point has been reached before, go to '**END OF INPUT RECORD SET**' (Step 20).

(17) Set RECLEN equal to -2.

(18) Enter *DETAB ENDFILE. (If no *DETAB ENDFILE is supplied in the program, the Compiler generates a *DETAB ENDFILE containing the single action IGNORE.)

(19) If return to the Fixed Logic is made by EXIT T, go to '**RECORD DECISION POINT**' (Step 4).

4.4.7.4 END OF INPUT RECORD SET

(20) If all records have been rejected at *DETAB RECORD, output the message 'NO DATA SELECTED', and terminate the run.

4.5 *TITLE Directive

As mentioned in *Printing Headings on page 19 of Chapter 2*, the *TITLE directive may be used to define a title to be given to a report. A title defined by this directive is normally printed by itself on an un-numbered page (that is, as a frontispiece to the report). Where a report title is not required to be a frontispiece, it should be defined using a *HEAD F directive (see **HEAD Directive on page 31*).

The *TITLE directive is optional; only one *TITLE directive may be included in a FILETAB program. Both *TITLE and *HEAD F directives may be included in a single FILETAB program.

4.5.1 Format of a *TITLE Directive

The general format is:

```
*TITLE  b,a
detail statements
```

4.5.1.1 The 'b' Operand

b specifies the paper control before the title is printed. The permitted value of b is CHn, where n is an integer in the range 0 to 63, and specifies at which line number on the first page the printing of the title is to begin. If this operand is omitted, CH0 is assumed.

4.5.1.2 The 'a' Operand

a specifies the paper control after the title is printed. The permitted values of a are CHn and n; in each case, n is an integer in the range 0 to 63. If this operand is omitted, a is assumed to equal 2.

If a is specified as n, the paper is advanced n lines after the title has been printed. If, however, this would cause page overflow, the paper is positioned at the top of the next page.

If a is specified as CHn, and the current printing position is at a line whose line number is not greater than n, the paper is positioned at line n on the current page after the title has been printed.

If a is specified as CHn, and the current printing position is at a line whose line number is greater than n, a throw to the top of the next page is performed, and the following steps is then taken:

If there is no *HEAD directive associated with the 'L' Control Level that has a b operand value of the form CHn (either explicitly or by default), the paper is positioned at the line on the new page specified by the *TITLE a operand.

If there is a *HEAD directive associated with the L Control Level that has a b operand value of the form CHn (either explicitly or by default), *LOOKUP BREAK L (if present) is performed and the *HEAD is printed. Printing of this heading begins at the line appropriate to the b operand on the *HEAD directive. After printing, the paper is positioned as specified by the a operand on the *HEAD directive. If the line required by the a operand on the *TITLE directive is lower down the page than this position, the paper is advanced to the lower line; otherwise, the paper remains at the position required by the *HEAD directive.

4.5.2 *TITLE Detail Lines

Each pair of detail lines specified on a *TITLE directive forms a single print line. A block of *TITLE detail lines is terminated by the next directive, see *Length of Lines on page 5 of Chapter 2* for further details.

4.5.3 Content of *TITLE Detail Lines

*TITLE detail lines may specify literal text. They may also include any of the following special strings:

```
DD/MM/YY   Current date
DD/MM/YYYY Current date
HH.MM.SS   Current time
PPPP       Page number
MKMK       Current version of FILETAB
```

Single quotes (') should not be used in *TITLE detail lines.

*Note: If PPPP is included in a *TITLE detail line, the value one is printed, and Reserved Word PAGE is then reset to zero. Whenever a new page is taken, PAGE is increased by one before any printing on the new page takes place.*

*If PPPP is specified both in *TITLE and in *HEAD L CHn detail lines, the Reserved Word PAGE should be set to one in *DETAB START. This is because *TITLE normally introduces an un-numbered frontispiece to a report; page numbering is specified in *HEAD L CHn, and begins from one on the page following the frontispiece.*

A *TITLE directive may not be included in a program if the directive *OPTION NOIN is present.

4.6 *HEAD Directive

As mentioned in *Printing Headings on page 19 of Chapter 2*, the *HEAD directive defines hierarchic headings to be produced throughout a report. Each *HEAD directive and its associated detail lines defines the format of a heading, and associates it with a particular control level and/or with the special L (List) control level.

The *HEAD directive is optional; more than one *HEAD directive may be included in a FILETAB program, but no more than one *HEAD may be associated with any particular control level.

4.6.1 Format of a *HEAD Directive

The general format is:

```
*HEAD  control-level(s)  b,a,p
detail statements
```

The following pages describe: firstly, the conditions under which a *HEAD is printed (the control-level(s) operand); secondly, paper control (the b, a and p operands); and thirdly, the content of headings ('detail lines').

4.6.1.1 The 'control-level(s)' Operand

This operand defines the control level or levels with which the specified heading is to be associated. Permitted values are either one of the control

characters M, N, O, P, Q, R and F, or the special List control level character L, optionally followed by one of the above control characters (for example L,M). The control character used must appear in the *INLIST. The operand may be omitted, if it is not required, in which case the *HEAD directive is associated with the L control level.

As mentioned above, no more than one *HEAD directive may be associated with any particular control level (except F or L).

Note: If F or L are coupled with another control level they must be seperated by commas with no spaces between.

The level or levels with which a heading is associated determine the conditions under which it is printed *(see the following descriptions)*.

4.6.2 Printing Order of Headings

Headings are printed during the first part of each control break cycle, that is, before the test to determine the level of the next control break. Headings associated only with control levels higher than the level of the current control break are not printed.

Apart from a heading associated with L (the special 'List' control level), all the headings in each control break cycle are printed in descending order of their associated control levels. A heading associated with the L level, either alone or in conjunction with another control level (for example, *HEAD L,M), is printed according to the following rules.

4.6.3 Printing *HEAD L

A heading associated with the L control level (either alone or in conjunction with another control level) is printed either on each new page taken (but not on the first page when *TITLE is used), or in every control break cycle, regardless of the level of the control break. Which alternative applies depends on the setting of the b operand on the *HEAD L directive. This operand is described below.

Where *HEAD L is to be printed on each new page, it is the first item to be printed. A typical application of this feature is the inclusion of the special string PPPP in the *HEAD L detail line, in order to obtain page numbering.

For example, if the following headings were included in a program:

```
*HEAD L CH2,2
 SALES REPORT FOR WEEK ENDING  DD/MM/YY   PAGE PPPP
*HEAD N CH4,2
 DIVISION  'NNNNNNNNNN'
*HEAD M 0,2
 AREA      'MMMMMMMMMM'



 SALESMAN  TOTAL SALES (£)
```

then, whenever a control break occurred at the N level, the following headings would be output:

```
line number
on page

0
1
2     SALES REPORT FOR WEEK ENDING  3/12/00   PAGE    1
3
4     DIVISION   NORTH WEST
5
6     AREA       MANCHESTER
7
8     SALESMAN   TOTAL SALES (£)
```

Note: The three blank detail lines of the `*HEAD M`; these represent the second half of the line beginning '`AREA`' and both halves of the following output line, which is required to be blank. (See Chapter 2.1.7 for further details.)

Where `*HEAD L` is to be printed, not on each new page, but in every control break cycle, all other headings to be printed during that cycle are printed first.

4.6.4 Special Rules for *HEAD L with another Control Level Specified

Where both the `L` control level and `M`, `N`, `O`, `P`, `Q`, `R` or `F` are associated with a single heading, as, for example:

```
*HEAD L,M
```

the heading is printed as if two separate `*HEAD` directives had been used (one at the `L` control level and one at the other specified control level), except that the heading is never printed twice on the same page within the same control break cycle.

4.6.5 Paper Control and Headings

Paper control before and after the printing of individual headings is specified by the `b` and `a` operands, respectively, on the `*HEAD` directive.

Note: It is important to note that the `b` and `a` operands specified on all heading directives are interrelated in two ways:

- Firstly, they must be consistent. If the paper is positioned either at the top of a page, or at the position reached after printing `*HEAD L CHn`, it must be possible to print, on the same page, the largest possible heading sequence (that is, `*HEAD` for the highest control level specified, down to `*HEAD` for the `M` level), without exceeding the maximum number of lines per page
- Secondly, a special rule applies to headings printed within a particular control break cycle (*see 'Special rule for headings', below*). This rule ensures that all such headings appear on one page, except for second and subsequent `*HEAD L CHn` output lines within the control break cycle

4.6.6 The ’b’ Operand

The b operand specifies paper control before printing. There are two alternative methods of paper positioning:

- 'Absolute' (by line number)
- 'Relative' (by the number of lines relative to the current printing position)

If b is omitted a value of CH0 is assumed.

4.6.6.1 ’Absolute’ Positioning

When the b operand is specified in the form CHn, where n is a line number, the paper is positioned at line n on the current page, unless this would be at, or higher up on the page than, the current line (in which case, the special rule described below applies). For example, if the paper is currently positioned at line 2 on a page, and the next item to be printed is:

```
*HEAD N CH4
```

the paper is positioned at line 4 before the heading is printed.

*Note: The first line on the page is line 0, not line 1, and that *HEAD L directives that have the CHn form of the b operand are printed only when a new page is taken.*

4.6.6.2 ’Relative’ Positioning

When the b operand is specified in the form n, the paper is advanced n lines before the heading is printed.

Unless the heading is the first to be printed within the current control break cycle, the paper is advanced, and the heading is printed on the current page. For example, if a *HEAD N has just been printed after a control break at the N control level, and the paper is currently positioned at line 20, the following *HEAD M directive would cause the paper to be advanced 3 lines:

```
*HEAD M 3,6
 AREA      'MMMMMMMMMM'
```

The one-line *HEAD M would thus be printed on line 23.

When the heading to be printed is the first within the current control break cycle, the paper is advanced, and the heading printed, either on the current page or on the next page, depending on the following conditions:

If the entire sequence of headings to be printed within the current control break cycle can be printed on the current page, the paper is advanced n lines on the current page, before the first heading in the sequence is printed, as in the above example. Otherwise, the paper is positioned at the top of the next page, *HEAD L CHn (if any) are printed, and the paper is advanced n lines.

*Note: All positioning specified on the *HEAD L CHn directive is obeyed first. For example, if a program contained the following:*

```
*HEAD M 1,6
 AREA    'MMMMMMMMMM'
*HEAD L CH2,2
 SALES REPORT FOR WEEK ENDING  DD/MM/YY   PAGE PPPP
```

and a control break at the `M` *level occurred, with the paper positioned too far down the page for the ensuing sequence of headings to fit, a new page would be taken. The headings on this new page would appear as follows:*

```
line on
page

0
1
2      SALES REPORT FOR WEEK ENDING  31/12/00   PAGE   27
3
4
5      AREA     MANCHESTER
6
```

4.6.7 Requesting New Pages at Certain Control Break Levels

It is possible to request that a new page be taken whenever a control break occurs at, or above, a certain control level. This is done by specifying `b` in the form `CHn` on all `*HEAD` directives specifying a control level greater than, or equal to, the required control level. `n` must have the same value on all these `*HEAD` directives.

Note: A new page is taken only if the printing position after a control break cycle is greater than the number of the line specified by `n`.

4.6.8 Special Rule for Headings

This rule is designed to prevent a sequence of headings from being split over more than one page; it is best illustrated by an example:

If the following headings were included in a program:

```
*HEAD L CH0,2
 SALES REPORT FOR WEEK ENDING  DD/MM/YY   PAGE PPPP
*HEAD N CH0,2
 DIVISION  'NNNNNNNNNN'
*HEAD M CH0,2
 AREA      'MMMMMMMMMM'



SALESMAN  TOTAL SALES (£)
```

then, whenever a control break occurred at the `M` level, or higher, a new page would be taken. (Again, three blank lines in the program represent the second half of the previous line and a single blank line, when output.) See *Length of Lines on page 5 of Chapter 2* for further details.

First, a control break at the `M` level is considered. In this case, a new page would be taken, and `*HEAD L` would be printed. This would be followed by `*HEAD M`, but, because the positioning requested before printing (that is, line 0) is not possible on the current page (since the paper can't be

moved backwards), and because a new page has already been taken to print the *HEAD L, the CH0 is replaced with a single line throw before the *HEAD M is printed.

Consequently, the output would look like this at the M break:

```
line number
on page

0      SALES REPORT FOR WEEK ENDING  31/12/00   PAGE   27
1
2
3      AREA       MANCHESTER
4
5      SALESMAN   TOTAL SALES (£)
6
7
```

Similarly, when a control break occurs at the N level, the CH0s requested before the printing of *HEAD M and *HEAD N would each be replaced by a single line throw. The output at the M break would consequently look like this:

```
line number
on page

0      SALES REPORT FOR WEEK ENDING  31/12/00   PAGE    1
1
2
3      DIVISION   NORTH WEST
4
5
6      AREA       MANCHESTER
7
8      SALESMAN   TOTAL SALES (£)
```

4.6.9 The ’a’ Operand

The a operand specifies paper control after printing. Only the ’relative’ method of paper positioning is permitted; that is, the paper is positioned by a number of lines relative to the current printing position. The number specified may be any non-negative integer. For example:

```
*HEAD L  CH2,4
```

*Note: Overprinting of a heading may be achieved by setting 'a' equal to '0', and setting the 'b' operand of the next print block (usually *OUT L) also to '0'.*

If the ’a’ operand is omitted, a value of ’2’ is assumed.

4.6.10 The ’p’ Operand

The p operand specifies the maximum size of pages in a report. It may be specified on any *HEAD or *OUT directive in the program. If it is specified on more than one directive, then the operand must be identical or a compilation error is generated.

Certain restrictions apply to the minimum value of p. If any of these restrictions is violated, the program run terminates with an error message. The restrictions are as follows:

(1) p must be such that *TITLE fits on the page, taking into account the paper control specified before the *TITLE is printed, but ignoring the paper control specified to follow the *TITLE.

(2) p must be such that the longest possible heading sequence fits on the page, taking into account the following:

- All paper control before the printing of each *HEAD in the sequence
- The printing of *HEAD L either before or after the sequence, as appropriate
- All paper control after the printing of each *HEAD in the sequence, except that specified to follow the last *HEAD

(3) p must be such that any individual *OUT fits on the page, preceded by *HEAD L CHn, if there is one, taking into account the following:

- The paper control required before and after printing the *HEAD L
- Any paper control specified to precede the *OUT

(4) The maximum permitted value of p is 99.

If p is omitted from all the *HEADs and *OUTs in a program, a default value of 60 is assumed.

4.6.11 Content of *HEAD Detail Lines

*HEAD detail lines are similar to those for the *TITLE directive, except that the contents of Control or Transfer fields currently held in the *INLIST may be printed. In order to be printed, strings of one or more copies of each FSC required must be enclosed within single quotes. For example, if the Control field M were defined in the *INLIST as a 12-character field and its current value were 'MANCHESTER ', then

```
*HEAD M
 AREA  'MMMMMMMMMMMMM'
```

would be printed as

```
 AREA   MANCHESTER
```

4.6.12 Substitution Rules for Character Control and Transfer Fields

Character Control and Transfer fields are treated as non-numeric, and the first FSC in a string is substituted with the first character from the field; this is known as 'left justification'. The rules for substitution follow:

(1) If the field length of a Control or Transfer field exceeds the length of the string of FSCs, it is truncated on the right. For example, if the

control field M were defined in the *INLIST as a 12 character field, and its current value were 'MANCHESTER ', then:

```
*HEAD M
 AREA  'MMMMM'
```

would be printed as:

```
 AREA   MANCH
```

(2) If the field length of a Control or Transfer field is less than that of the string of FSCs, substitution occurs in a cyclic manner. For example, if the control field 'M' were defined in the *INLIST as a 12 character field, and its current value were 'MANCHESTER ', then:

```
*HEAD M
 AREA  'MMMMMMMMMMMMMMMM'
```

would be printed as:

```
 AREA   MANCHESTER  MANC
```

(3) If more than one string occurs for a particular FSC, either in the same or different detail lines, substitution continues in each string from the point at which it ceased in the previous string. For example, if the control field M were defined in the *INLIST as a 12 character field, and its current value were 'MANCHESTER ', then:

```
*HEAD M
 AREA  'MMMM'       'MMMM'
```

would be printed as:

```
 AREA   MANC         HEST
```

(4) Unless it is used as an FSC, any of the editing symbols described under '*Editing symbols within numeric fields' in Chapter 4.7.14* may be freely used as an editing symbol within character strings for output by the *HEAD directive. For example, if the control field M were defined in the *INLIST as a 12 character field, and its current value were 'MANCHESTER ', then:

```
*HEAD M
AREA  'M.M.M.M.M.M.M.M.M.M.M.M.'
```

would be printed as:

```
AREA   M.A.N.C.H.E.S.T.E.R. . .
```

(5) Special strings (for example, DD/MM/YY) may not be enclosed within pairs of single quotes. Similarly, literal text should not be enclosed within quotes, since any character which appears as an FSC in the *INLIST is substituted.

4.6.13 Substitution Rules for Binary Control and Transfer Fields

Binary Control and Transfer fields are treated as numeric, and are converted to character form before substitution for an FSC string. The rules for substitution are the same as those for Totalling fields in *OUT detail lines (*see *OUT Directive on page 39*).

4.6.14 Transfer Fields Printed in Headings

When a Transfer field is printed in a heading, its value is taken from the first detail record of the control group for the control level to which the heading applies.

4.7 *OUT Directive

The *OUT directive defines hierarchic output print blocks to be produced throughout a report. Each *OUT directive and its associated detail lines defines the format of an output block, and associates it with one or more of the control levels L, M to R and F.

The *OUT directive is optional; more than one *OUT directive may be included in a FILETAB program, but no more than one *OUT may be associated with any particular control level. Where no *OUT directive is specified, a default *OUT L is supplied, containing, in its detail lines, an FSC string for each FSC defined in the *INLIST. Control fields are printed in descending order of FSCs. All other fields are printed in ascending order of FSCs. For character Control and Transfer fields, the length of the string output is the same as the length defined in the *INLIST; for numeric Transfer and Totalling fields, an allowance of 12 characters is made for each string. Each field is separated from the next by two spaces.

4.7.1 Format of a *OUT Directive

The general format is:

```
*OUT  control-level(s)  b,a,p
detail statements
```

The following pages describe: firstly, the conditions under which a *OUT is printed (the 'control-level(s)' operand); secondly, paper control (the b, a and p operands); and thirdly, the content of output blocks ('detail lines').

4.7.2 The 'control-level(s)' Operand

This operand defines the control level or levels with which the specified output block is to be associated. Permitted values are any or all of the Control Field Specifying Characters (that is, L, M to R and F). Where more than one character is specified, the characters must be separated by commas.

As mentioned above, no more than one *OUT directive may be associated with any particular control level.

The level or levels with which an output block is associated determine the conditions under it is printed, *(see the following descriptions)*.

4.7.3 Printing Order of Output Blocks

Output blocks are printed during the second part of each control break cycle, that is, after the test to determine the level of the next control break.

In each control break cycle, output blocks are printed in ascending order of their associated control levels, beginning with the `L` level. Where an output block is associated with more than one control level, it is printed once for each associated control level, but not for those levels higher than the level of the newly-determined control break.

4.7.4 Paper Control and Output Blocks

Paper control before and after the printing of individual output blocks is specified by the `b` and `a` operands, respectively, on the `*OUT` directive.

4.7.5 The ’b’ Operand

The `b` operand specifies paper control before printing. There are two alternative methods of paper positioning:

- 'absolute' (by line number)
- 'relative' (by the number of lines relative to the current printing position)

4.7.6 ’Absolute’ Positioning

When the `b` operand is specified in the form `CHn`, `n` refers to a line number on the current page or the next page, as follows.

If the line specified is further down the same page, the positioning specified refers to the current page; otherwise, it refers to the next page. When positioning refers to the next page, `*HEAD L CHn` (if any) is printed on the new page before the `*OUT` is printed.

The line specified by `n` must be such that it is possible to print the output block, beginning at line `n`, without exceeding the maximum number of lines per page specified by the `p` operand (*which is discussed below*).

4.7.7 ’Relative’ Positioning

When the `b` operand is specified in the form `n`, the paper is either left at the current line, or positioned `n` lines further down the same page; `n` represents a non-negative integer, in the range 0 to 63. If `n` is omitted, a default of 1 is assumed.

For each individual output block, a test is performed, to decide whether it is possible to obey the `b` operand and print the entire output block without exceeding the maximum number of lines per page (*see the `p` operand, described below*). If the maximum number of lines per page would be exceeded, a new page is taken, any `*HEAD L` with a `b` operand of the form `CHn` is printed, the `b` operand on the `*OUT` directive is obeyed, and the output block is printed.

4.7.8 The 'a' Operand

The a operand specifies paper control after printing, and operates like the b operand, except in one respect.

When a is specified in its absolute form, requesting that the paper be positioned at a line higher up on the next page, and the program contains a *HEAD L directive with a b operand in the form CHn, positioning may be affected in one of two ways:

(1) If the line number specified by the a operand on the *OUT directive is not greater than the line number specified by the b operand on the *HEAD L directive, the paper is, in effect, positioned on the next page at the line specified by the a operand on the *OUT directive, and then advanced to the line specified by the b operand on the *HEAD L directive, at which point the *HEAD L is printed.

(2) If the line number specified by the a operand on the *OUT directive is greater than the line number specified by the b operand on the *HEAD L directive, the *HEAD L is printed on the next page at the line specified by the b operand on the *HEAD L directive. The paper is then, in effect, advanced by the number of lines specified by the a operand on the *HEAD L directive, and then advanced to the line number specified by the a operand on the *OUT directive.

*Note: In this case, the line number specified by the a operand on the *OUT directive must be sufficiently large to allow such advancement after the printing of the *HEAD L (that is, without the need for a further new page).*

4.7.9 The 'p' Operand

The p operand specifies the maximum size of pages in a report. It may be specified on any *HEAD or *OUT directive in the program. If it is specified on more than one directive, then the p operand must be identical otherwise a compilation error is generated.

Certain restrictions apply to the minimum value of p. If any of these restrictions is violated, the program run terminates with an error message. The restrictions are as follows:

(1) p must be such that *TITLE fits on the page, taking into account the paper control specified before the *TITLE is printed, but ignoring the paper control specified to follow the *TITLE.

(2) p must be such that the longest possible heading sequence fits on the page, taking into account the following:

- All paper control before the printing of each *HEAD in the sequence
- The printing of *HEAD L either before or after the sequence, as appropriate
- All paper control after the printing of each *HEAD in the sequence, except that specified to follow the last *HEAD

(3) p must be such that any individual *OUT fits on the page, preceded by *HEAD L CHn, if there is one, taking into account the paper control required before and after printing the *HEAD L, and any paper control specified to precede the *OUT.

If p is omitted from all the *HEADs and *OUTs in a program, a default value of 60 is assumed.

4.7.10 Content of *OUT Detail Lines

As well as literal text, *OUT detail lines may contain special strings and strings of Control, Transfer and Totalling FSCs.

An important difference between *OUT and *HEAD detail lines is that, in *OUT detail lines, literal text and special strings must appear within pairs of single quotes, and FSC strings must not appear in single quotes.

Example

```
*HEAD L CH0,2
   SALES REPORT FOR WEEK ENDING  DD/MM/YY       PAGE PPPP
*HEAD N CH0,2
   DIVISION 'NNNNNNNNNNNN'
*HEAD M CH0,2
   AREA     'MMMMMMMMMMM'



   SALESMAN                              TOTAL SALES (£)

*OUT L 1,0
   AAAAAAAAAAAAAAA                         11,111,111.11
*OUT M 2,0
  'SALES TOTAL FOR' MMMMMMMMMMM 'AREA'     11,111,111.11
*OUT N 2,0
  'SALES TOTAL FOR' NNNNNNNNNNNNN 'DIVISION'11,111,111.11
```

A typical page from this report would look like this:

```
line number
on page

0  SALES REPORT FOR WEEK ENDING  31/12/00       PAGE   27
1
2
3  DIVISION  NORTH WEST
4
6  AREA      MANCHESTER
7
8  SALESMAN                              TOTAL SALES (£)
9
10
11 BRIGGS J.                                       12.47
12 CROSS K.                                        13.12
13 ELDRITCH P.                                    382.83
14 WHATMOUGH A.                                   139.24
15
16 SALES TOTAL FOR  MANCHESTER   AREA             547.66
17
```

```
18 SALES TOTAL FOR  NORTH WEST    DIVISION       9,443.11
```

4.7.11 Literal Text and Special Strings in *OUT Detail Lines

These are substituted just as they are in `*HEAD` detail lines.

4.7.12 Control and Transfer Field FSC Strings

These are substituted just as they are in `*HEAD` detail lines, with the following important difference. When a transfer field is printed in an output line, its value is taken from the last detail record of the control group for the control level to which the output line applies.

4.7.13 Totalling Field FSC Strings

When Totalling fields are moved from the Record Area to the `*INLIST` Area, they are changed from the format defined under the `*INLIST` directive into single-word (that is, 32-bit) signed binary values, and are held thus in the `*INLIST` Area, see `*ALTER FSCT`, *Dummy Totalling Fields on page 4*.

Before a Totalling field is substituted in an `*OUT` detail line, its cumulative value, appropriate to the control level at which the `*OUT` is being printed, is accessed, and converted into a number in character form; zero is converted into a single zero digit. After this conversion, the FSCs in the string are replaced, from right to left, by the corresponding digits of the number.

If there are more digits in the number than FSCs in the string, the entire FSC string is replaced with asterisks.

If there are fewer digits in the number than FSCs in the string, the remaining portion of the field is substituted according to one of three options as follows:

4.7.13.1 *OPTION ZS - Leading Zero Suppression

The un-substituted portion of the field, including editing symbols (*see below*) is overwritten with spaces, except that any unsubstituted FSC that immediately precedes a dot or a space, and all FSCs that follow a dot or a space, are replaced by zeros. Floating symbols are then dealt with (*see below*).

4.7.13.2 *OPTION CZS - Complete Zero Suppression

This is the same as zero suppression except when applied to zero Totalling fields. For these, the FSC string and preceding floating symbols associated with it are completely overwritten with spaces.

4.7.13.3 *OPTION NZS - No Leading Zero Suppression

The unsubstituted FSCs in the string are replaced with zeros. All editing symbols are printed as they appear.

4.7.14 Editing Symbols within Numeric Fields

The following symbols may be used to edit strings of numeric FSCs, provided that they themselves have not been used as FSCs. These symbols are of two types:

- Floating Symbols `£ $ * + -`
- Editing Symbols `/,.` and `space`

4.7.15 Floating Symbols

Up to three of the symbols `£$*+-` may precede a totalling FSC string. These symbols 'float' past leading spaces in the substituted field.

`£` and `$` have the normal meaning of 'pounds' and 'dollars'.

`*` is known as the 'cheque protect' symbol, and can be used, in conjunction with £ or $, to fill in any space left between the `£` or `$` sign and an actual amount (for example, `£**111.11`).

+ and - are used to pre-sign totalling fields.

Note: + is not printed if a field pre-signed with it has a negative value. Conversely, - is not printed if a field pre-signed with it has a positive value.

4.7.16 Editing Symbols

The symbols `/,.` and `space` may be used within the string of FSCs. The symbols `/,` and `.` may also immediately precede or follow a string of FSCs. Any number of these symbols may be used.

Note: If two editing symbols appear adjacent to each other, the FSC string is treated as two separate strings, the second editing symbol being assumed to belong to the second string.

4.7.17 Examples of Editing and Floating Symbols

In the following example, three totalling fields `TOT1`, `TOT2` and `TOT3` are associated with the FSCs 1, 2 and 3. At the `L` level, for the record currently in the `*INLIST` Area, `TOT1`, `TOT2` and `TOT3` contain the binary values 0, 6 and -3125, respectively.

```
*PROGRAM EXAMPLE
*OPTION ZS
*
*DICTIONARY
 TOT1 = 4:4
 TOT2 = 8:4
 TOT3 = 12:8
```

```
*
*INLIST
 1 TOT1,2 TOT2,3 TOT3
*
*OUT L 1,0
 £111111     £*2,222,222.22

  +11111     +22222      +33333

  -11111     -22222      -33333

 $*+2222     1111..1111   3333//3333
```

After substitution, the output would be as follows

```
      £0     £*********0.06
       0         +6        3125
       0          6       -3125
 $****+6        0..0000         3125
```

4.7.18 Post-signing of Totalling Fields

It is possible to suffix a Totalling field, according to its sign, with two characters. These characters are specified either on `*OPTION PS` (for non-negative fields) or `*OPTION NS` (for negative fields).

Note: Where post-signing is required, two spaces must be left for the two characters; if insufficient space is allowed, no post-signing takes place.

4.8 Printer Control Characters (Format Effectors)

Note: Only those who may wish to override the normal printer control provided by FILETAB need study this section.

Each FILETAB output line is prefixed, by the Fixed Logic, with a printer control character. The printer control character is called `PP0` *(see Chapter 4.2.15)*. It can be used in two different ways:

- To hold a negative value causing positioning at a specific line on the page
- To hold a positive value causing positioning at a line relative to the current line on the page (this must not be used in the first line printed by the program)

Where a printer control character specifies positioning at a particular line number, a page throw is implied if the current line number is greater than or equal to that specified by the printer control character. In other words, positioning at either the current line, or a line higher up the page, automatically causes a page throw. In this case, `PPPP` is automatically updated.

Note: 'Top of Form' is line zero therefore -1 in `PP0` is equal to line zero and -2 in `PP0` is equal to line 1 and so on.

Function	`PP0` Contents
Top of Form	`-1`
Skip to line 10	`-11`
Skip to line 16	`-17`
Single-line Feed	`1`
Advance 10 lines	`10`
Advance 16 lines	`16`

Within a `*HEAD` printing sequence, only one page throw may be implied. Where a second page throw would be implied, the Fixed Logic overrides the specified printer control, and spaces two lines before printing.

By default, the Fixed Logic automatically generates the required printer control characters, using the information provided on `*HEAD`, `*OUT` and `*TITLE` directives.

Multi-line `*HEAD, *OUT` and `*TITLE` print blocks are normally printed with single-line spacing, unless an alternative format is specified in `*DETAB PRINT`.

Chapter 5
Program Control

The following features are described in this chapter:

- Naming the program
- Compiler listing control (compiler listing *OPTIONs)
- Other complier *OPTIONs
- Program run-time *OPTIONs
- Incorporation of external source
- Termination of the program
- Formatting the compilation listing

5.1 Naming the Program

The program is named during compilation. The compiled program assumes the name of the source program. For further details of compilation see *Appendix A*.

5.2 * OPTIONs

The *OPTION directive controls source printing, complier behaviour, report editing and some run-time functions. The format of *OPTION is as follows:

```
*OPTION    operand(s)
```

5.2.1 Compiler Listing *OPTIONs

The following operands are used to control the printing of compiler listings:

NP	Do not print FILETAB source statements during compilation
P	Print FILETAB source statements during compilation. This *OPTION is selected by default
PDS	Print source statements double-spaced during compilation
PSS	Print source statements single-spaced during compilation. This *OPTION is selected by default
PST	Print symbol table. This is a diagnostic aid, and is fully described in *Appendix A*
XREF	Print cross reference listing. This is a diagnostic aid, and is fully described in *Appendix A*

*OPTIONs PST and XREF have a global effect. The other *OPTIONs affect only the statements which follow them. Any *OPTIONs which are required to affect the whole set of source statements should appear immediately after the *PROGRAM directive.

More than one option may be specified on a single *OPTION directive, but they must be separated by commas with no intervening spaces:

```
*OPTION  PDS,PST
```

*Note: *OPTION PST does not work if NP is used.*

If no *OPTION directive is specified, FILETAB assumes the following default:

```
*OPTION DR,NK,P,PSS
```

Details of *OPTIONs DR and NK appear in *Program Run-time *OPTIONs on page 3*.

5.2.2 Other Compiler *OPTIONs

ALI/NALI	If *OPT ALI is set then Binary, Unsigned, Floating Point and Internal Decimal fields which do not have a specified start address will be aligned on word boundaries (if they are a multiple of a word in length). This is useful when using file types which insist on these items being aligned (such as INFORMIX SQL) and will improve performance. If *OPT NALI is set no alignment occurs. Default NALI
CL	Specifies that the code may contain continuation lines *(*see *Length of Lines on page 5* of *Chapter 2)*
CORE	If a run time error is reported will cause a 'core dump' to be produced - this is useful for debugging purposes. Default No Dump Produced. Global
INTOPT/ DFLOPT	Set the *OPT values at the start of execution to those appearing before the first *DETAB / the default values (this is the default)
NCL	Specifies that the code does not contain continuation lines *(*see *Length of Lines on page 5* of *Chapter 2)*
NOIN	This option must be present if the source statements do not contain an *INLIST. Global
PE	After compilation, do not execute the program if there are source errors. Global
PL=integer	Specifies the length of the print line in bytes. If the CL option (above) is current then the line length is equal to half the integer value *(*see *Length of Lines on page 5* of *Chapter 2)*
PW	After compilation, do not execute the program if there are source warnings. Global
STDC/NSTDC	Controls the kind of C produced by the FILETAB Compiler STDC ANSI C NSTDC Classic C (this is useful for systems where Classic C is the default, for example, SUN OS4). Default STDC. Global

5.2.3 Program Run-time *OPTIONs

The following *OPTIONs control editing and run-time functions for FILETAB programs

BME	Binary Move Error. Generates a run time error if a binary field is moved to a location not big enough to take it. BME is set as default
BMF	Binary Move Fatal. Terminates the run when a program error is returned. BMF is set as default

CCAD	Align CC1 etc. with record position 0 (rather than with the standard position 4). This is to allow FILETAB to map both native files and those converted from other formats (usually defined as *FILEs) on to the start of the Record Area. By default, CC1 is defined as 4/1
CZS	Completely suppress leading zeros. A totalling field containing only zero is printed as spaces. When a binary field containing zero is moved to a character field, the character field contains spaces after the operation
DF	Controls the format of the date fields, (see *Appendix B*)
DP	Specifies decimal places in floating point to character moves, (see *Appendix B*)
DR	Divide rounded; for example, 3 divided by 2 gives a quotient of 2. The Reserved Word REMAINS (*see Reserved Words on page 1 of Chapter 7*) holds the remainder after division. In the example quoted, it contains -1. This *OPTION is selected by default
DU	Divide unrounded; for example, 3 divided by 2 gives a quotient of 1. The Reserved Word REMAINS holds the remainder. In the example quoted, it contains 1
DW	Controls the date window for two digit years, (see *Appendix B*)
DZE	Divide by zero error. When a number is divided by zero, a divide by zero error occurs. This *OPTION is selected by default
DZU	Divide by zero unchanged. When a number is divided by zero, the quotient is unchanged
DZZ	Divide by zero giving zero. When a number is divided by zero, the quotient is zero
FSCT = integer	Sets the default Totalling FSCs (unchanged by *ALTER) to use the type specified. The type numbers are: 1 Single Word binary (:4), 2 Double Word Binary (:8), 5 Floating Point ($8), 10 Internal Decimal (:24<INT_DEC>). Default 1 (Single Word Binary)
FSCT1	Specifies the default size for totalling fields as one word (:4) (see **ALTER Directive on page 2* of *Chapter 4*)
FSCT2	Specifies the default size for totalling fields as two words (:8) (see **ALTER Directive on page 2* of *Chapter 4*)
GI	Group Indicate. Print control FSCs only once they have changed or after a new page. *Note: avoid overwriting Control FSCs / IGNORE in *DET PRINT*
IS	Ignore Sign. When moving a character field to a binary field ignore the numeric sign in the character. IS is set as default
K	From a READ of *TABLE, return both key and descriptor

NBME	No Binary Move Error. No run time error is generated if a Binary field is moved to a location not big enough to take it. BME is set as default
NBMF	No Binary Move Fatal. The run is not terminated when a program error is returned. BMF is set as default
NIS	No Ignore Sign. When moving a character field to a binary field, include the numeric sign in the character. IS is set as default
NK	From a READ of *TABLE, return descriptor alone. This *OPTION is selected by default
NOC	No Output Comment. All program code lines after a *OUT, *HEAD, *PICT or *TITLE (up to the next directive) are treated as detail lines
NOLSS	No OutLine Special String (default mode). Special strings that are normally recognised in the environment, for example DD/MM/YY for date and PPPP for page number, are NOT to be recognised by FILETAB within *OUT literals. If NOLSS is set then DD/MM/YY is treated as a string 'DD/MM/YY' and not as a date
NRFIL	No Record Fill spaces (default mode). Specifies that records being read from the main file which are shorter than the record length are NOT to be padded
NSxx	Post-sign negative totalling fields, by printing the characters represented by xx after each occurrence of a negative value *Note that a field containing zero is regarded as positive in this context*
NTSZ	No Trailing Spaces as Zeros. When moving a character field to a binary field, trailing spaces are NOT to be treated as zeros. TSZ is set as default
NZS	No Zero Suppression. Do not suppress leading zeros. A totalling field is printed with leading zeros. When a binary field is moved to a character field, the character field contains leading zeros, if space permits
OC	Output Comment. If the last details line for a *OUT, *HEAD, *PICT or *TITLE starts with an * and a space, the line is treated as a comment. (This is for compatibility with UNITAB)
OLSS	OutLine Special String. Special strings that are normally recognised in the environment, for example DD/MM/YY for date and PPPP for page number, are to be recognised by FILETAB within *OUT literals
PSxx	Post-sign positive totalling fields, by printing the characters represented by xx after each occurrence of a positive value. A field containing zero is regarded as positive in this context

RB	Retained Binary. This replaces ':' with ';' in non-word binary field definitions. (This does not affect field definitions that use indirect length.) Fields that are changed in this way appear as ';' on Symbol Table reports
RFIL=xx	Record Fill spaces. Specifies that records being read from the main file which are shorter than the record length are to be padded with trailing characters of hex. code xx. Global
RFILS	Record Fill spaces. Specifies that records being read from the main file which are shorter than the record length are to be padded to the specified length with trailing spaces. Global
RPNx	Reset the page number to 1 on change of control group represented by x
SFIL / NFIL	Affects the moving of a shorter character field to a longer one: SFIL The longer field is padded with trailing spaces. NFIL The extra portion of the longer field (to the right) is left unchanged. Default SFIL
SMS=integer	Sort Memory Size. Specifies the amount of virtual memory in bytes allocated to sorts. The default size is 16Mb
TSZ	Trailing Spaces as Zeros. When moving a character field to a binary field, trailing spaces are to be treated as zeros. TSZ is set as default
ZS	Suppress leading zeros. A totalling field containing only zero is printed with leading spaces and one zero. When a binary field containing zero is moved to a character field, the character field contains leading spaces and one zero

Apparently contradictory *OPTIONs may appear in the same set of source statements. For instance, the two *OPTIONs DU (divide unrounded) and DR (divide rounded) might be used in a single program, when the application requires that some divisions are rounded and some not. In this example, the first division is unrounded, and the second rounded:

```
*OPTION DU
*DETAB ONE
A  SALESA  /  QTYA
*OPTION DR
*DETAB TWO
A  SALESB  /  QTYB
```

More than one option may be specified on a single *OPTION directive, but they must be separated by commas:

```
*OPTION DR,PDS,PST
```

If no *OPTION directive is specified, FILETAB assumes the following default:

```
*OPTION DR,ZS,NK,P,PSS,DZE,BME,BMF,IS,TSZ,NALI,NOLSS,NRFIL,
SFIL
```

5.3 Incorporating External Source

FILETAB programs normally consist of a series of statements contained in a single file, but it is also possible to form a program from sets of statements held in several files, by means of the *COPY directive.

5.3.1 *COPY Directive

*COPY allows the inclusion, in the main source stream, of parameters from other source files. The format is:

```
*COPY filename
```

Nesting of *COPY filename directives is allowed.

When a subsidiary parameter file is exhausted, the parameter following the *COPY which invoked it is read.

Any part of a set of FILETAB source statements may be included by the use of *COPY. For instance, if several programs use the same file, the dictionary for that file might reside on a subsidiary parameter file.

For example, COPFILE1 contains the following dictionary for FILEA:

```
FAFLD1  = 200/10
FAFLD2  = 210:8
FAFLD3  = 218:4
...etc
FAFLD20 = 392/8
```

and the main source file contains:

```
*PROGRAM PROG1
*FILE    MAINFILE  RL=200  NAME=mainfile.txt
*IFILE   FILEA     RL=200
*DICTIONARY
 MFFLD1 = 0/1
 MFFLD2 = 1/3
 MFFLD3 = 4:4
*COPY COPFILE1
*DETAB RECORD
...etc
```

Here, the compiler reads the main source file until it encounters the *COPY directive, at which point it reads the statements held in COPFILE1. After reading the last statement in COPFILE1 (FAFLD20 = 392/8), it returns to the main source file, and read the *DETAB RECORD statement.

*COPY may be usefully employed to include any set of statements (decision tables, dictionaries etc.) which perform a task common to two or more applications.

A portion of source included by means of the *COPY directive may itself contain a further *COPY directive. Nesting of *COPY directives is allowed.

5.4 Terminating the Program

A set of FILETAB source statements is terminated by `*STOP`, `*GO` or `*END`. Any of these directives indicates that the source program is complete.

If `*STOP` is specified, syntax-checking of the source takes place, and an object program is produced, unless there are syntax errors and `*OPTION` PE is included in the set of source statements.

The effect of `*GO` is the same as that of `*STOP`. `*GO` is provided merely for compatibility with 'load-and-go' versions of FILETAB.

If `*END` is specified, only syntax-checking of the source takes place. No object program is produced.

(See *Appendix A* for details on compiling.)

5.5 Formatting the Compilation Listing

The `*PAGE` directive is used only during the printing of source statements. Its appearance in a set of source statements causes the `*PAGE` directive to be printed at the top of the next page of the compiler listing. The format is:

```
*PAGE  comment
```

Chapter 6
Files, Tables

6.1 The Main Input File

The facilities already described enable reports to be produced from a single file. This chapter describes how:

- More than one input file may be accessed
- Output files may be accessed
- Indexed-Sequential and Direct files may be accessed and updated
- Lookup tables may be formed and accessed

6.1.1 *FILE

The `*FILE` directive defines the main file, from which the report is to be produced. Records are read automatically from the main file into the Record Area by the Fixed Logic.

Each set of FILETAB source statements must contain a `*FILE` directive, unless all processing is being performed at `*DETAB START` (*see Chapter 4.2.8*), or Reserved Word `RECLEN` is maintained at a value less than -1 (*see Chapter 4.2.10*).

Only one `*FILE` may be defined; subsidiary files are accessed by the `*IFILE`, `*OFILE`, `*EFILE` and `*IOFILE` directives. The format of the `*FILE` directive is:

```
*FILE  localname  keyword1=value1,keyword2=value2,...etc.
NAME=physicalname
```

'`localname`' can be of unlimited length and may contain hyphens. It must however, begin with an alphabetic character. It is used to link the logical file in the program with the physical file (as defined in either the `NAME` keyword or the file environment variable of the same name).

A keyword may not appear more than once. The list of keyword/values may be split over more than one input line, if the last value on each split line is terminated by a comma. However, a keyword/value pair must not be split. A keyword may be preceded by one or more spaces, but the keyword/value pair may not contain spaces. Character strings given as

values are normally abbreviated to their first character. For example, 'LINE' is given as 'L'. Possible values for the keyword/value pairs follow.

File name	
keyword: acceptable value: default:	NAME a file name none (see note below)

The NAME keyword associates the localname specified with the physical filename in value.

Note: An alternative method is available which allows the physical filename to be specified in an environment variable.

Filenames explicitly specified using NAME have precedence over any in environment variables.

If neither method is used a runtime error will be produced.

The RL keyword affects the number of bytes transferred into the data area.

Record length	
keyword: acceptable value: default:	RL a positive integer 80

The maximum length of records on the file is assumed to be that specified by the Record Length keyword (RL). The value of RL specifies the maximum number of bytes transferred from the file. It is important to remember that, if RL is defaulted, the maximum amount transferred is 80 bytes.

The OR keyword defines the organisation of the file, and operates in conjunction with the file directive (*FILE in this instance) to define the file handling verbs that are used to access it (*see Chapter 6.6*).

Directive	Organisation	Access allowed
*FILE	LINE (OR=Line) INDEXED-SEQUENTIAL (OR=I) DIRECT (OR=D)	READ READ, LOOKUP READ, LOOKUP

Because the FILETAB Fixed Logic performs automatic READs from the main file, the only explicit file access verb which may be legitimately issued is LOOKUP, and this is possible only if the file is Indexed-Sequential or Direct (*see 'LOOKUP verb and *FILEs', below*).

For Indexed-Sequential files, the position and length of the key may be defined by PK.

Key position	
keyword: acceptable value: default:	PK positive integer/positive integer for example, 9/9. 0/4 (This indicates that the key starts in byte 0 and has a length of 4 bytes).

6.1.1.1 Examples

```
*FILE INFILE1 OR=I,PK=4/4   NAME=infile1.idx
                            [4 bytes of data precede the key
*FILE INFILE2 OR=I,PK=1/10  NAME=infile2.idx
```

Note: When the length of the record is less than that specified on the RL *keyword, the record is transferred in its entirety, and the remainder of the area reserved for it remains undefined. The reserved word* RECLEN *contains the actual number of bytes transferred.*

6.1.2 LOOKUP Verb and *FILEs

It is possible to use this verb with Indexed-Sequential and Direct *FILEs at the START or RECORD Decision Point. See *LOOKUP Verb and *IOFILEs on page 20* for further details of LOOKUP on files.

6.1.2.1 Notes

When the end of the main file is reached, it is closed automatically. If there is a *DETAB ENDFILE in the program, it is entered. Further details appear in *ENDFILE Decision Point on page 15 of Chapter 4*.

It is possible to close the main file early, by setting the Reserved Word RECLEN to zero or -1 at the RECORD Decision Point. Further details appear in *Use of RECLEN at the RECORD Decision Point on page 11 of Chapter 4*.

6.1.3 Use of FF, VV and SUBCOUNT

If the main file contains variable length records in sub-record format (that is, where each record consists of a fixed length portion followed by a variable number of fixed length sub-records), processing can be simplified by use of the keywords FF and VV, specified on the *FILE directive. Use of these keywords enables FILETAB to process each sub-record as a separate record. This means that special decision tables are not required to ’unpack’ files held in this way.

Reserved Words FF and VV may be accessed by decision tables, to vary the values defined on the *FILE directive.

When FF and VV are declared on a *FILE directive, FILETAB reads the fixed portion of the record, and the first sub-record. The next read simply overwrites the first sub-record with the second sub-record, and so on, until

they have all been presented. The following example illustrates the use of FF and VV, and the function of SUBCOUNT, which contains the number of the sub-record within the current record, starting from zero. The physical file contains:

the fixed portion	sub-record A	sub-record B	sub-record C

After the first READ, the Record Area contains:

the fixed portion	sub-record A	Reserved Word SUBCOUNT = 0

After the second READ, the Record Area contains:

the fixed portion	sub-record B	Reserved Word SUBCOUNT = 1

After the third READ, the Record Area contains:

the fixed portion	sub-record C	Reserved Word SUBCOUNT = 2

The next READ transfers the fixed portion of the next physical record, and its first sub-record, into the Record Area. When a record on the file consists of a fixed portion without sub-records, the fixed portion is transferred to the Record Area as normal, and the sub-record portion of the Record Area is filled with binary zeros and SUBCOUNT is set to -1.

It is possible to cause the next READ to access the next physical record, by moving the value -1 to SUBCOUNT. The current contents of the Record Area are unchanged until the next READ occurs.

The length in bytes of the fixed portion of a record is defined initially by the FF keyword. This may be altered by a decision table at the RECORD level of processing, which moves a value into the Reserved Word FF.

The length in bytes of the variable portion of a record is defined initially by the VV keyword. Again, this may be amended by a decision table at the RECORD level.

The Reserved Words FF and VV are initialised with the *FILE values only at the start of the run, so it is important to ensure that they continue to hold the correct values after amendment. When a record on the file is smaller than FF, the whole record is moved to the Record Area, the rest of the fixed portion and the sub-record portion of the Record Area are filled with binary zeros and SUBCOUNT is set to -2.

When a record on the file has a variable portion which is not exactly divisible by VV, the fixed portion is moved to the Record Area, the sub-record portion of the Record Area is filled with binary zeros and SUBCOUNT is set to -3.

6.1.4 Closing The Main File Early

It is possible to close the main file before end-of-file has been reached. The method is described in Further details appear in *Closing The Main File Early on page 16 of Chapter 4*.

6.2 Other Input File

Any number of subsidiary input files may be read, and are defined by the *IFILE directive. Records are read into the Record and Work Areas when a READ verb is issued from a decision table.

6.2.1 *IFILE Directive

```
*IFILE localname keyword1=value1,keyword2=value2,...etc
NAME=physicalname
```

'localname' can be of unlimited length and may contain hyphens. It must however, begin with an alphabetic character. It is used to link the logical file in the program with the physical file (as defined in either the NAME keyword or a file environment variable of the same name). It is also used as an operand with READ, LOOKUP and CLOSE verbs.

A keyword may not appear more than once. The list of keyword/values may be split over more than one input line, if the last value on each split line is terminated by a comma. However, a keyword/value pair must not be split. A keyword may be preceded by one or more spaces, but the keyword/value pair may not contain spaces. Character strings given as values are normally abbreviated to their first character. For example, 'YES' is given as Y. Possible values for the keyword/value pairs are given below.

File name	
keyword: acceptable value: default:	NAME a file name none (see note below)

The NAME keyword associates the localname specified with the physical filename in value.

Note: An alternative method is available which allows the physical filename to be specified in an environment variable.

Filenames explicitly specified using NAME have precedence over any in environment variables.

Note: If neither method is used a runtime error will be produced.

Keywords which affect the number of bytes transferred into the data area are BC (Byte Count) and RL (Record Length). The transfer may also be

influenced by the form of the READ statement used. The READ statement is discussed later in this section.

Byte count	
keyword: acceptable value: default:	BC YES, NO, Y, N NO

Record length	
keyword: acceptable value: default:	RL a positive integer 80

The BC keyword indicates whether a count of the number of bytes involved in each transfer is to be placed in the first four bytes of the buffer. If the count is requested (BC=Y), it is used only by the software, and is not transferred from the file. The count does not include the four bytes holding the count.

Regardless of the value associated with BC, the length of the record is assumed to be either that specified by the Record Length keyword (RL), or that specified by the READ statement. Precedence is given to that specified by the READ, unless this exceeds the value given on RL.

The value of RL specifies the maximum number of bytes that can be transferred from the file. It is important to remember that, if RL is defaulted, the maximum amount transferred is eighty bytes.

6.2.1.1 Examples

```
*IFILE INFILE1 BC=Y,RL=100  NAME=infile.txt
*IFILE INFILE2 RL=500,BC=N  NAME=infile.txt
*IFILE INFILE3 RL=20    NAME=infile.txt  [default BC=NO
*IFILE INFILE4 BC=Y     NAME=infile.txt  [default RL=80
*IFILE INFILE5         NAME=infile.txt  [default RL=80,BC=NO
```

The OR keyword defines the organisation of the file, and operates in conjunction with the file directive (*IFILE in this instance) to define the file handling verbs that are used to access it (*see Chapter 6.6*).

Directive	Organisation	Access allowed
*IFILE	LINE (OR=Line) INDEXED-SEQUENTIAL (OR=I) without alternate index INDEXED-SEQUENTIAL (OR=I) with alternate index ENVIRONMENT (OR=J) DIRECT (OR=D)	READ READ, LOOKUP READ,INDEX,LOOKUP READ READ, LOOKUP

The ENVIRONMENT organisation and its use are described in Chapter 6.7.

For Indexed-Sequential files, the position and length of the key may be defined by PK.

Key Position	
keyword: acceptable value: default:	PK positive integer/positive integer for example, 9/9 0/4 (This indicates that the key starts in byte 0 and has a length of 4 bytes).

6.2.1.2 Examples

```
*IFILE INFILE1 OR=I,PK=4/4   NAME=infile.idx
                            [4 bytes of data precede the key
*IFILE INFILE2 OR=I,PK=1/10  NAME=infile.idx
```

For Direct files the key does not form part of the record, but is an integer referring to a particular position within the file. Thus PK is redundant.

Indexed-Sequential files may be accessed by an alternate index. Each alternate index must be defined using AK*x*=*y*/*z* (where x equals an integer determining the alternate index number, y is the key position and z is the key length). If duplicate keys are permitted for an alternate index it must also be specified for each index using ADx=Y (where x equals an integer determining the alternate index number).

6.2.1.3 Examples

```
*IFILE INFILE1 OR=I,PK=4/1   NAME=infile.idx AK1 = 20/4
                    [ The file has an alternate index and
                    [ the major key at byte four for one byte
*IFILE INFILE3 OR=I,PK=1/10  NAME=infile.idx
                    [ no alternate index.
```

6.2.2 READ Verb and *IFILEs

The READ verb causes a record to be read from a file which has been defined by an *IFILE directive. Its format is:

```
READ   localname   data-area
```

'localname' is as defined on the associated *IFILE directive.

'data-area' defines a field into which the data is to be read. It may be any valid character Field Definition, the name of a field already defined in the *DICTIONARY, or, at a BREAK Decision Point only, an FSC preceded by an asterisk (*FSC).

For example, if the *IFILE directive for INFILE contains BC=NO (either explicitly or as a default), any of the READ statements in the following example would read the first fifty bytes from the next record in the named file, transferring the data into character 100 onwards of the Record Area.

```
*DICTIONARY
 BUFFER  = 100/50
 BUFFER2 = 100/80

*DETAB READFILE
A  READ INFILE 100/50     [ THESE STATEMENTS ARE
   READ INFILE BUFFER     [   EXACTLY
   READ INFILE BUFFER2/50 [   EQUIVALENT
```

If the `*IFILE` directive contains `BC=YES`, 46 bytes of data are read into character 104 onwards. The first four bytes of the data-area contain the number of bytes transferred (forty six in this example).

It is possible to `READ` specifying a length of zero, for example:

```
READ  INFILE  100/0
```

In this case, FILETAB examines the relevant `*IFILE` directive and, if it contains `BC=YES`, the record length is taken from the value of the `RL` keyword. The number of bytes transferred is placed into the first four bytes of the data-area (100:4 in this example). The first data character of the record is placed into character 104 of the Record Area.

If the `*IFILE` directive contains `BC=NO`, the number of bytes transferred is equal to the value of the `RL` keyword. The first data character of the record is placed into character 100 of the Record Area.

The interaction of the `READ` statement with the `*IFILE` directive is best illustrated by the following table:

	READ statement length non-zero	**READ statement length zero**
BC=YES or BC=Y	Record is read into start-of-data address+4, and length is data-area length minus four bytes	Record is read into start-of data address+4, and length is that specified by the RL keyword
BC=NO or BC=N	Record is read into start-of-data address, and length is that specified on the READ statement	Record is read into start-of-data address, and length is that specified by the RL keyword

The `READ` verb may be used in both limited and extended entry decision tables.

The start-of-data address plus the length, stated directly or indirectly, must not exceed the Record or Work Area limits.

When end-of-file is reached following a `READ`, the first four bytes of the buffer area are set to X'FFFFFFFF' (signed binary -1). This result should be tested and confirmed, after each `READ` action. A further `READ` re-opens the file, and positions it as though it had not been accessed previously.

When the record is smaller than the area reserved for it, the record is transferred in its entirety, and the remainder of the reserved area remains

undefined. Where appropriate, the first four bytes of the data-area contain the actual number of bytes transferred.

The BC keyword is not necessary with Direct files, as they contain fixed length records.

When a record is read from an Indexed-Sequential file with an alternate index, the reserved word ZREPLY returns the following values:

Value	Meaning
0	Record read successfully
-101	The next record within the key of reference has the same value as that key (this only occurs in a file that allows duplicates)

6.2.3 LOOKUP Verb and *IFILEs

It is possible to use this verb with Indexed-Sequential files with an alternate index. *See Input/Output Files on page 14* for further details of the use of INDEX and LOOKUP on files.

It is possible to use this verb with Direct and Indexed-Sequential *IFILEs. See *Input/Output Files on page 14* for further details of LOOKUP on files.

6.2.4 Closing *IFILEs

An input file may be closed before the end of the run by the issue of a CLOSE verb. This procedure is necessary only when it is required to close a file prematurely. Its format is:

```
CLOSE   localname
```

’localname’ is as defined on the associated *IFILE directive.

Any further references to a CLOSEd file cause the file to be re-opened and positioned as though it had not been accessed previously.

6.3 Printer Output File

Each FILETAB program contains a built-in *OFILE (*see Chapter 6.4*) called the Printer Output file. This file has a localname of FTC_REPORT. A particular filestore file may be specified for use as FTC_REPORT. If the specified file does not exist, it is created.

The Fixed Logic writes records to the Printer Output file as a result of *TITLE, *HEAD and *OUT detail lines.

6.4 Other Output Files

Any number of output files may be accessed; they are defined by the *OFILE or *EFILE directive. Records are constructed, usually in the record and work areas, using decision tables, and written to an output file by the WRITE verb. The difference between *OFILE and *EFILE is that, whereas *OFILE clears the output file and writes records to it from the beginning, *EFILE always appends records to the existing output file.

6.4.1 *OFILE/*EFILE Directives

```
*OFILE)  localname keyword1=value1,keyword2=value2,...etc
*EFILE)                    NAME=physicalname
```

'localname' can be of unlimited length and may contain hyphens. It must however, begin with an alphabetic character. It is used to link the logical file in the program with the physical file (as defined in either the NAME keyword or a file environment variable of the same name). It is also used as an operand with WRITE, CLOSE and (on *OFILE only) LOOKUP verbs.

A keyword may not appear more than once. The list of keyword/values may be split over more than one input line, if the last value on each split line is terminated by a comma. However, a keyword/value pair must not be split. A keyword may be preceded by one or more spaces, but the keyword/value pair may not contain spaces. Character strings given as values are normally abbreviated to their first character. For example, 'YES' is given as 'Y'. Possible values for the keyword/value pairs are given below.

File name	
keyword: acceptable value: default:	NAME a file name none (see note below)

The NAME keyword associates the localname specified with the physical filename in value.

Note: An alternative method is available which allows the physical filename to be specified in an environment variable.

Filenames explicitly specified using NAME have precedence over any in environment variables.

Note: If neither method is used a runtime error will be produced.

Record size is affected by the keywords BC (Byte Count) and RL (Record Length). Record size may also be influenced by the form of the WRITE statement used. *The WRITE statement is discussed later in this section.*

Byte count	
keyword: acceptable value: default:	BC YES, NO, Y, N NO

Record length	
keyword: acceptable value: default:	RL a positive integer 80

The BC keyword indicates whether a count of the number of bytes involved in each transfer is located in the first four bytes of the buffer.

The count, if requested (BC=Y), is used only by the software, and is not transferred to the file. The count does not include the four bytes holding the count.

If no count is present (BC=N), the length of the record is assumed to be either that specified by the Record Length keyword (RL), or that specified by the WRITE statement.

Precedence is given to that specified by the WRITE, unless this exceeds the value given on RL.

The value of RL specifies the maximum number of bytes that can be transferred to the file. It is important to remember that, if RL is defaulted, the maximum record size is eighty bytes.

6.4.1.1 Examples

```
*OFILE OUTFILE1 BC=Y,RL=100  NAME=outfile.txt
*OFILE OUTFILE2 RL=500,BC=N  NAME=outfile.txt
*EFILE OUTFILE3 RL=20   NAME=outfile.txt     [default BC=NO
*EFILE OUTFILE4 BC=Y    NAME=outfile.txt     [default RL=80
*OFILE OUTFILE5         NAME=outfile.txt     [default
                                             [RL=80,BC=NO
```

The OR keyword defines the organisation of the file, and operates in conjunction with the file directive (*OFILE/*EFILE in this instance) to define the file handling verbs that are used to access it (*see Chapter 6.6*).

Directive	Organisation	Access allowed
*OFILE	LINE (OR=Line) INDEXED-SEQUENTIAL (OR=I) DIRECT (OR=D)	WRITE WRITE, LOOKUP WRITE, LOOKUP

Directive	Organisation	Access allowed
*EFILE	LINE (OR=Line) INDEXED-SEQUENTIAL (OR=I) DIRECT (OR=D)	WRITE WRITE WRITE

Because *EFILE is used to WRITE in append mode, the file organisation may not be or Index Sequential or Direct.

The Environment organisation (OR=I) and its use are described in *Accessing Environment Variables on page 24*.

For Indexed-Sequential the position and length of the key may be defined by PK.

Key position	
keyword: acceptable value: default:	PK positive integer/positive integer for example, 9/9 0/4 (This indicates that the key starts in byte 0 and has a length of 4 bytes).

6.4.1.2 Examples

```
*OFILE OUTFILE1 OR=I,PK=4/4  NAME=outfile.idx
                        [4 bytes of data precede the key
*OFILE OUTFILE2 OR=I,PK=1/10 NAME=outfile.idx
```

For Direct files, the key does not form part of the record, but is an integer referring to a particular position within the file. Thus PK is redundant.

6.4.2 WRITE Verb and *OFILEs/*EFILEs

The WRITE verb causes a record to be written to a file which has been defined by an *OFILE/*EFILE directive. Its format is:

```
WRITE   localname   data-area
```

'localname' is as defined on the associated *OFILE/*EFILE directive.

'data-area' defines a field from which the data is to be written. It may be any valid character Field Definition, the name of a field already defined in the *DICTIONARY, or, at a BREAK Decision Point only, an FSC preceded by an asterisk (*FSC).

For example, if the *OFILE/*EFILE directive for OUTFILE contains BC=NO (either explicitly or as a default), any of the following WRITE statements would write a fifty byte record to the named file, taking the data from character 100 onwards in the Record Area.

```
*DICTIONARY
 BUFFER  = 100/50
 BUFFER2 = 100/80
```

```
*DETAB WRITEFIL
A  WRITE OUTFILE 100/50      [ THESE STATEMENTS ARE
   WRITE OUTFILE BUFFER      [   EXACTLY
   WRITE OUTFILE BUFFER2/50  [   EQUIVALENT
```

If the *OFILE/*EFILE directive contains BC=YES, forty six bytes of data are written from character 104 onwards. The four bytes holding the byte count are not written to the file, and four bytes are subtracted from the data length specified. It is possible to WRITE specifying a length of zero. For example:

```
WRITE  OUTFILE  100/0
```

In this case, FILETAB examines the relevant *OFILE/*EFILE directive and, if it contains BC=YES, the correct record length, in binary, is assumed to be in the first four bytes of the data-area (100:4 in this example). The length is not written to the output file, so the first data character of the record is character 104 of the Record Area.

If the *OFILE/*EFILE directive contains BC=NO, a record whose length is equal to the value of the RL keyword is written. The interaction of the WRITE statement with the *OFILE/*EFILE directives is best illustrated by the following table

	WRITE statement length non-zero	**WRITE statement length zero**
BC=YES or BC=Y	Record is written from start-of-data address+4, and length is data-area length minus four bytes	Record is written from start-of-data address+4, and length is to be found in first four bytes of data-area
BC=NO or BC=N	Record is written from start-of-data address, and length is that specified on the WRITE statement	Record is written from start-of-data address, and length is that specified by the RL keyword

The WRITE verb may be used in both limited and extended entry decision tables.

The start-of-data address plus the length, either directly or indirectly stated, should not exceed the Record or Work Area limits.

The BC keyword is not necessary for Direct files, as they contain fixed length records.

6.4.3 Closing *OFILEs/*EFILEs

An output file may be closed before the end of the run by the issue of a CLOSE verb. This procedure is necessary only when it is required to close a file prematurely. Its format is:

```
CLOSE   localname
```

'localname' is as defined on the associated *OFILE/*EFILE directive.

Any further references to a CLOSEd Line file cause the file to be re-opened and positioned either at the start of file (for *OFILEs) or at the end of file (for *EFILEs).

Any further references to a CLOSEd Indexed-Sequential or Direct file cause the file to be re-opened and positioned as though it had not been accessed previously.

6.5 Input/Output Files

Any number of Indexed-Sequential or Direct files may be accessed and updated, and are defined by the *IOFILE directive. The verbs READ, WRITE, LOOKUP, DELETE and CLOSE are available for file-handling.

6.5.1 *IOFILE Directive

The format of the *IOFILE directive is:

```
*IOFILE localname keyword1=value1,keyword2=value2,...etc
NAME=physicalname
```

'localname' can be of unlimited length and may contain hyphens. It must however, begin with an alphabetic character. It is used to link the logical file in the program with the physical file (as defined in either the NAME keyword or the file environment variable of the same name). It is also used as an operand with READ, WRITE, LOOKUP, DELETE and CLOSE verbs.

A keyword may not appear more than once. The list of keyword/values may be split over more than one input line, if the last value on each split line is terminated by a comma. However, a keyword/value pair must not be split. A keyword may be preceded by one or more spaces, but the keyword/value pair may not contain spaces. Character strings given as values are normally abbreviated to their first character. For example, 'YES' is given as 'Y'. Possible values for the keyword/value pairs follow.

File name	
keyword: acceptable value: default:	NAME a file name none (see note below)

The NAME keyword associates the localname specified with the physical filename in value.

Note: An alternative method is available which allows the physical filename to be specified in an environment variable.

Filenames explicitly specified using NAME have precedence over any in environment variables.

Note: If neither method is used a runtime error will be produced.

Record size is affected by the keywords BC (Byte Count) and RL (Record Length). Record size may also be influenced by the form of the file-handling verb used. These verbs are discussed later in this section.

Byte count	
keyword: acceptable value: default:	BC YES, NO, Y, N NO

Record length	
keyword: acceptable value: default:	RL a positive integer 80

The BC keyword indicates whether the first four bytes of the buffer are to hold a count of the number of bytes involved in each operation.

The count, if requested (BC=Y), is used only by the software, and is not transferred to or from the file. The count does not include the four bytes holding the count.

If no count is present (BC=N), the length of the record is assumed to be either that specified by the Record Length keyword (RL), or that specified by the file-handling verb. Precedence is given to that specified by the verb, unless this exceeds the value given on RL.

The value of RL specifies the maximum number of bytes that can be read from or written to the file. It is important to remember that, if RL is defaulted, the maximum record size is eighty bytes.

6.5.1.1 Examples

```
*IOFILE  FILE1 BC=Y,RL=100  NAME=file1.txt
*IOFILE  FILE2 RL=500,BC=N  NAME=file2.txt
*IOFILE  FILE3 RL=20    NAME=file3.txt [default BC=NO
*IOFILE  FILE4 BC=Y     NAME=file4.txt [default RL=80
*IOFILE  FILE5          NAME=file5.txt [default RL=80,BC=NO
```

The OR keyword defines the organisation of the file and operates in conjunction with the File directive (*IOFILE in this instance) to define the file handling verbs that are used to access it (*see Chapter 6.6*).

Directive	Organisation	Access allowed
*IOFILE	INDEXED-SEQUENTIAL (OR=I) without alternate index INDEXED-SEQUENTIAL (OR=I) with alternate index DIRECT (OR=D)	READ, WRITE, LOOKUP, DELETE READ, WRITE, INDEX, LOOKUP, DELETE READ, WRITE, LOOKUP

For input/output files, the position and length of the key may be defined by PK.

Key Position	
keyword: acceptable value: default:	PK Positive integer/positive integer for example, 9/9 0/4 (this indicates that the key starts in byte 0 and has a length of four bytes).

6.5.1.2 Examples

```
*IOFILE  FILE1 OR=I,PK=4/4   NAME=file1.idx
                        [4 bytes of data precede the key
*IOFILE  FILE2 OR=I,PK=1/10  NAME=file2.idx
```

For Direct files, the key does not form part of the record, but is an integer referring to a particular position within the file. Thus PK is redundant.

Indexed-Sequential files may be accessed by an alternate index. Each alternate index must be defined using AK*x*=*y*/*z* (where x equals an integer determining the alternate index number, y is the key position and z is the key length). If duplicate keys are permitted for an alternate index it must also be specified for each index using ADx=Y (where x equals an integer determining the alternate index number).

6.5.1.3 Examples

```
*IOFILE  FILE1 OR=I,PK=4/1,AKI=20/4,ADI=Y  NAME=file1.idx
                   [ The file has an alternate index and
                   [ the major key at byte four for one byte.
*IOFILE  FILE2 OR=I,PK=4/1  NAME=file2.idx
                                       [no alternate index
```

6.5.2 WRITE Verb and *IOFILEs

The WRITE verb causes a record to be written to a file which has been defined by an *IOFILE directive. Its format is:

```
WRITE   localname   data-area
```

'localname' is as defined on the associated *IOFILE directive.

'data-area' defines a field from which the data is to be written. It may be any valid character Field Definition, the name of a field already defined in the *DICTIONARY, or, at a BREAK Decision Point only, an FSC preceded by an asterisk (*FSC).

For example, if the *IOFILE directive for FILE contains BC=NO (either explicitly or as a default), any of the following WRITE statements would write a fifty byte record to the named file, taking the data from character 100 onwards in the Record Area.

```
*DICTIONARY
 BUFFER  = 100/50
 BUFFER2 = 100/80

*DETAB READFILE
A  READ INFILE 100/50     [ THESE STATEMENTS ARE
   READ INFILE BUFFER     [   EXACTLY
   READ INFILE BUFFER2/50  [   EQUIVALENT
```

If the *IOFILE directive contains BC=YES, forty six bytes of data are written from character 104 onwards. The four bytes holding the byte count are not written to the file, and four bytes are subtracted from the data length specified.

It is possible to WRITE specifying a length of zero, for example:

```
WRITE  FILE  100/0
```

Here, if the relevant *IOFILE directive contains BC=YES, the correct record length, in binary, is assumed to be in the first four bytes of the data-area (100:4 in this example).

The length is not written to the output file, so the first data character of the record is character 104 of the Record Area.

If the *IOFILE directive contains BC=NO, a record is written whose length equals the value of the RL keyword.

The interaction of the WRITE statement with the *IOFILE directive is best illustrated by the following table:

	WRITE statement length non-zero	**WRITE statement length zero**
BC=YES or BC=Y	Record is written from start-of-data address+4, and length is data-area length minus four bytes	Record is written from start-of data address+4, and length is to be found in the first four bytes of the data-area
BC=NO or BC=N	Record is written from start-of-data address, and length is that specified on the WRITE statement	Record is written into start-of-data address, and length is that specified by the RL keyword

WRITE may be used in both limited and extended entry forms.

Start-of-data address plus length, directly or indirectly stated, should not exceed the Record or Work Area limits.

Both the record upon which the WRITE verb acts, and the file position into which the record is written, are dependent on the previous file-handling verb used. Full details may be found later in this chapter.

The BC keyword is not necessary for Direct files, as they contain fixed length records.

When a record is written to an Indexed-Sequential file with an alternate index, the reserved word ZREPLY returns the following values:

Value	Meaning
0	Record written successfully.
-100	Record written has an alternate key that is equal to a record already on the file (this only occurs on files where duplicates are allowed).

6.5.3 READ Verb and *IOFILEs

The READ verb reads a record from a file defined by an *IOFILE directive. Its format is:

```
READ  localname  data-area
```

'localname' is as defined on the *IOFILE directive.

'data-area' defines a field into which the data is to be read. It may be any valid character Field Definition, the name of a field already defined in the *DICTIONARY, or, at a BREAK Decision Point only, an FSC preceded by an asterisk (*FSC).

For example, if the *IOFILE directive for INOUT contains BC=NO (either explicitly or as a default), any of the following statements would read the first 50 bytes from the next record in the named file, transferring the data into character 100 onwards of the Record Area.

```
*DICTIONARY
  BUFFER  = 100/50
  BUFFER2 = 100/80

*DETAB READFILE
A  READ INOUT 100/50      [ THESE STATEMENTS ARE
   READ INOUT BUFFER      [   EXACTLY
   READ INOUT BUFFER2/50   [   EQUIVALENT
```

If the *IOFILE directive contains BC=YES, 46 bytes of data are read into character 104 onwards. The first four bytes of the data-area contain the number of bytes transferred (46 in this example).

It is possible to READ specifying a length of zero:

```
READ   INOUT   100/0
```

Here, the relevant *IOFILE directive is examined and, if it contains BC=YES, the record length is taken from the value of the RL keyword. The number of bytes transferred is placed into the first four bytes of the data-area (100:4 in this example). The first data character of the record is placed into character 104 of the Record Area.

If the *IOFILE directive contains BC=NO, the number of bytes transferred is equal to the value of the RL keyword. The first data character of the record is placed into character 100 of the Record Area.

The interaction of the READ statement with the *IOFILE directive is best illustrated by the following table:

	READ statement length non-zero	READ statement length zero
BC=YES or BC=Y	Record is read into start-of-data address+4, and length is data-area length minus four bytes.	Record is read into start-of data address+4, and length is that specified by the RL keyword
BC=NO or BC=N	Record is read into start-of-data address, and length is that specified on the READ statement.	Record is read into start-of-data address, and length is that specified by the RL keyword.

The READ verb may be used in both limited and extended entry decision table lines.

Start-of-data address plus length, directly or indirectly stated, should not exceed the Record or Work Area limits.

When end-of-file is reached following a READ, the first four bytes of the buffer area are set to X'FFFFFFFF' (binary -1). This result should be tested and confirmed after each READ action.

The actual record in the file which is read depends on the preceding file-handling verb. A full description of this interaction may be found later in this chapter.

The BC keyword is not necessary for Direct files, as they contain fixed length records.

When the length of the record is less than that of the area reserved for it, the record is transferred in its entirety, and the remainder of the reserved area remains undefined. Where appropriate, the first four bytes of the data-area contains the actual number of bytes transferred.

When a record is read from an Indexed-Sequential file with an alternate index, the reserved word ZREPLY returns the following values:

Value	Meaning
0	Record read successfully.

Value	Meaning
-101	The next record within the key of reference has the same value as the key (this only occurs in a file that allows duplicates).

6.5.4 LOOKUP Verb and *IOFILEs

LOOKUP should be used only with Direct and Indexed-Sequential files. LOOKUP searches by key, and indicates whether a record with a particular key already exists. Its format is:

```
LOOKUP   localname   data-area
```

LOOKUP, unlike the other file-handling verbs, is a condition verb, and must have a condition entry supplied. A Y (yes) means that the key is present (a 'true' LOOKUP); an N (no) means that it is not present (a 'false' LOOKUP).

'localname' is as defined on the associated *IOFILE directive.

'data-area' defines a character field holding the key of the record which is sought. It may be any valid character field definition, the name of a field already defined in *DICTIONARY, or, at a BREAK decision point only, an FSC preceded by an asterisk (*FSC). In the case of a direct file, the data area must be a :4 field.

For example, if a value of 1234 had been moved into 100/4, the condition statement:

```
C  LOOKUP   ISFILE   100/4     Y   N
```

would determine whether the key 1234 was present on the file associated with local name ISFILE.

If it were, any conditions and actions below the Y would be performed (a 'true' LOOKUP). If it were not, any conditions and actions below the N would be performed (a 'false' LOOKUP).

The interaction between LOOKUP *and other file-handling verbs is discussed more fully later in this chapter.*

6.5.5 INDEX Verb and *IOFILEs

INDEX should only be used with Indexed-Sequential files that have an alternate index. INDEX determines which alternate index LOOKUP and READ search by. Its format is:

```
INDEX  localname keynumber
```

'localname' is as defined in the *IOFILE directive.

'keynumber' defines the number of the key to use as defined on the *IOFILE directive. Key number 0 (zero) refers to the main (primary) key.

For example, if alternate index 2 starts at byte 73 of the record and is four bytes long, and a value of 1234 is moved into the field to be used for the alternate index.

```
I  IREC+73/4 MV '1234'
   INDEX ISFILE 2
C  LOOKUP ISFILE IREC+73/4   Y  N
```

The above FILETAB statements would determine whether the key 1234 was present on the file associated with the name ISFILE. If it were, any conditions and actions below the Y would be performed (a 'true' LOOKUP). If it were not, any conditions and actions below the N would be performed (a 'false' LOOKUP).

The interaction between INDEX and other file-handling verbs is discussed in more detail later in this chapter.

6.5.6 DELETE Verb and *IOFILEs

The DELETE verb causes a record to be removed from an Indexed-Sequential file. Its format is:

```
DELETE   localname
```

'localname' is as defined on the associated *IOFILE directive.

The record deleted is the current record; this is the last record to have been accessed by a true LOOKUP or by a READ statement.

A DELETE should not be issued after a false LOOKUP, since the record deleted in such an instance is indeterminate.

A DELETE statement cannot be actioned on a Direct file. It is not possible to remove records, only to nullify them, for example, space-fill.

6.5.7 Closing *IOFILEs

An *IOFILE may be closed before the end of the run by the issue of a CLOSE verb. This is necessary only when it is required to close a file prematurely. The format is:

```
CLOSE   localname
```

'localname' is as defined on the associated *IOFILE directive.

Any further references to a CLOSEd file re-opens it.

6.6 Interaction of Verbs and File Organisation

The following lists indicate the interaction between, and limitations on, file handling verbs and file organisation.

Directive	Organisation	Access allowed

`*FILE`	`LINE (OR=Line)`
	`INDEXED-SEQUENTIAL (OR=I)`
	`DIRECT (OR=I)`
		`READ`
		`READ, LOOKUP`
		
		`READ, LOOKUP`

`*IFILE`	`LINE (OR=Line)`
	`INDEXED-SEQUENTIAL (OR=I)` without alternate index
	`INDEXED-SEQUENTIAL (OR=I)` with alternate index
	`ENVIRONMENT (OR=J)`
	`DIRECT (OR=D)`
		`READ`
		`READ, LOOKUP`
		
		`READ,INDEX,LOOKUP`
		
		`READ`
		`READ, LOOKUP`

`*OFILE`	`LINE (OR=Line)`
	`INDEXED-SEQUENTIAL (OR=I)`
	`DIRECT (OR=D)`
		`WRITE`
		`WRITE, LOOKUP`
		
		`WRITE, LOOKUP`

`*EFILE`	`LINE (OR=Line)`
	`INDEXED-SEQUENTIAL (OR=I)`
	`DIRECT (OR=D)`
		`WRITE`
		`WRITE`
		
		`WRITE`

`*IOFILE`	`INDEXED-SEQUENTIAL (OR=I)` without alternate index
	`INDEXED-SEQUENTIAL (OR=I)` with alternate index
	`DIRECT (OR=D)`
		`READ, WRITE, LOOKUP, DELETE`
		
		`READ, WRITE, INDEX, LOOKUP, DELETE`
		`READ, WRITE, LOOKUP`

Note: `CLOSE` access is given automatically on all files other than the Main File (`*FILE`). The Main File is closed automatically at the end of a run; closing the Main File early is described in Closing The Main File Early on page 16 of Chapter 4.

6.6.1 Serial Files (OR=L)

`READ` may follow only `READ`.

`WRITE` may follow only `WRITE`.

6.6.2 Indexed-Sequential Files (OR=I)

`READ` following `READ`, `WRITE` or `DELETE` accesses the next record in key sequence.

`READ` following `LOOKUP` (TRUE) accesses the record with the key specified in the `LOOKUP`.

`READ` following `LOOKUP` (FALSE) accesses the first record with a key higher than that specified in the `LOOKUP`.

`READ` following `INDEX` accesses the record with the lowest key in that index.

`WRITE` following `LOOKUP` (TRUE) overwrites the record with the key specified in the `LOOKUP`. The key of the new record must be identical to that specified in the `LOOKUP`.

`WRITE` following `LOOKUP` (FALSE) inserts a new record at the appropriate point in the file. The key of the record to be inserted must be the same as that specified in the `LOOKUP`. It must be neither greater than the highest key on the file, nor smaller than the lowest key on the file.

`WRITE` following `READ` overwrites the record accessed by the `READ`. The key of the record to be written must be the same as that in the record just read.

`DELETE` following `LOOKUP` (TRUE) destroys the record with the key specified in the `LOOKUP`.

`DELETE` following `READ` destroys the record accessed by the `READ`.

`LOOKUP` may follow `READ`, `WRITE` or `DELETE`.

`WRITE` may not follow `WRITE` or `DELETE`.

`DELETE` may not follow `WRITE`, `DELETE` or `LOOKUP` (FALSE).

6.6.3 Direct Files (OR=D)

`READ` following `READ` or `WRITE` reads the next record.

`READ` following `LOOKUP` (TRUE) reads the record whose key was specified in the `LOOKUP`.

`READ` following `LOOKUP` (FALSE) is not supported for Direct files.

`WRITE` following `LOOKUP` (TRUE) overwrites the record whose key was specified in the `LOOKUP`.

`WRITE` following `LOOKUP` (FALSE) is not supported for Direct files.

`WRITE` following `READ` overwrites the record accessed by the `READ`.

`DELETE` is not supported for Direct files.

`LOOKUP` may follow `READ` or `WRITE`.

`WRITE` may not follow `WRITE`.

6.7 Accessing Environment Variables

Data may be transferred from environment variables. This is done by issuing a `READ` verb with the appropriate `localname`. The corresponding `FILE` directive must also be present in the program, as below:

```
*IFILE localname  OR=J  NAME=physicalname
```

'`localname`' is the name given to the environment variable.

File name	
keyword: acceptable value: default:	NAME a file name none (see note below)

The `NAME` keyword associates the `localname` specified with the physical filename in value.

Note: An alternative method is available which allows the physical filename to be specified in an environment variable.

Filenames explicitly specified using `NAME` have precedence over any in environment variables.

Note: If neither method is used a runtime error will be produced.

The format of the necessary `READ` or `WRITE` statement is:

```
READ      localname   field
```

Where the `localname` is declared as an environment variable, '`field`' may be any valid character field definition.

6.8 Tables

As described in *Tables Accessed by the Fixed Logic on page 17* of *Chapter 4*, tables may be created within sets of source statements, to associate keys with descriptors. Any number of tables may be created in a set of source statements. A character field may be compared with a table, to test for the presence of a particular key, or to access the descriptor associated with the key, or to do both.

Two directives are used to define tables: `*LOOKUP` and `*TABLE`. When `*LOOKUP` is used, FILETAB accesses the table at the required decision point, automatically. When `*TABLE` is used, the point at which access is made, can be controlled. `*LOOKUP` is described in **LOOKUP Directives on page 18* of *Chapter 4.*

6.8.1 *TABLE Directive

The `*TABLE` directive defines a `LOOKUP` table. The format is:

```
*TABLE   tablename   size
entry format statement
text statements
```

’tablename’ can be of unlimited length and may contain hyphens. It must however, begin with an alphabetic character.

The size parameter is no longer required and if specified, is ignored. It has been included purely for compatibility with other versions of FILETAB.

’entry format statement’ defines the position of keys and descriptors on the following ’text statements’, by means of Ks and Ds. Each K represents one character of key, and each D represents one character of descriptor. The Ks must always precede the Ds. The total number of Ks and Ds must not exceed 80 characters. Spaces must not be embedded within the group of Ks or the group of Ds, but may appear between the two groups. ’Key-only’ tables are permissible; a fuller description appears later in this chapter.

Examples of valid and invalid entry format statements follow.

Valid	Invalid
KKKDDD KKK DDD KKK	K KKDDD KKKD DD DDDKKK DDD

’text statements’ follow the entry format statement, and specify the contents of the table entries. The position of the text must correspond to that specified on the entry format statement. If, when the table is being compiled, the keys are not presented in sequence, the entries are sorted into ascending order. Duplicate keys are accepted as valid entries, but the sequence of such keys is indeterminate. Any number of text statements may be specified.

Here is an example of a *TABLE:

```
*TABLE  COLOURS
K DDDDDD
1 RED
2 BLUE
4 GREEN
6 PUCE
```

Tables are processed by the verbs READ, WRITE, LOOKUP, SELECT, DELETE and CLOSE, each of which is described separately below. Examples of access to *TABLE COLOURS follow these descriptions.

6.8.2 READ Verb and *TABLEs

READ is an action verb that retrieves an entry from a table. The precise effect of its use depends on the most recent action performed on the table.

If the first action performed on a table is a READ, the descriptor accessed corresponds to the lowest key value in the table.

Successive READ actions retrieve entries in key sequence. If a READ is performed after the highest key entry, an end-of-table descriptor (X'FF') is read. If a subsequent READ is performed, the table is then read from the lowest key onwards.

Note: A LOOKUP searching for a value of X'FF' naturally returns 'TRUE', because of the presence of this 'end-of-table' descriptor.

A READ following a false LOOKUP, WRITE or DELETE causes the descriptor with the next higher key to be retrieved. If such a record does not exist, the end-of-table descriptor (X'FF') is read.

A READ following a true LOOKUP or SELECT causes retrieval of the descriptor corresponding to the key value used in the LOOKUP or the entry SELECTed. The format of READ is:

```
READ   tablename   data-area
```

'tablename' is the name by which the table is known.

'data-area' defines a field into which the data is to be read. It may be either any valid character Field Definition, the name of a field already defined in the *DICTIONARY, or, at a BREAK decision point only, an FSC preceded by an asterisk (*FSC).

For example, if AREA1 were defined in the *DICTIONARY as 100/50, the following READs on TABLEA would have identical effects:

```
A  READ TABLEA AREA1
   READ TABLEA 100/50
   READ TABLEA AREA1/50
```

Either the key and descriptor, or the descriptor alone, is read, depending on the setting of *OPTION K or NK (*see Chapter 5.2.3*).

If the data-area is of a different length from the key and descriptor, or descriptor alone, it is either truncated or space-filled, as required.

If a READ action encounters the end of the table, the whole of the data area is set to X'FF's.

Note: Examples of the use of table-handling verbs appear at the end of this chapter.

6.8.3 WRITE Verb and *TABLEs

WRITE is an action verb that adds an entry to a table, or overwrites an existing entry.

A WRITE following a false LOOKUP adds an entry, in the correct sequence, to a table. A WRITE following a true LOOKUP or SELECT replaces the entry found by the LOOKUP or the entry SELECTed. When a WRITE follows a LOOKUP, the key of the record to be written must be the same as the key specified in the LOOKUP.

WRITE must never be the first action performed on a table; it must always be preceded by a LOOKUP, READ or SELECT.

When a WRITE follows a READ, and the new key is less than the existing one, an entry is inserted, provided this is not out of key sequence. If the keys are equal, the existing entry is overwritten. If the new key is greater than the existing one and/or out of key sequence, a program event is caused.

WRITE should never follow WRITE. The format of WRITE is:

```
WRITE   tablename   data-area
```

'tablename' is the name of a table previously defined on a *TABLE directive.

'data-area' defines a field that contains the entry to be written to the table. It may be any valid character Field Definition, or the name of a field already defined in the *DICTIONARY. For example, if AREA1 were defined in the *DICTIONARY as 100/50, the following WRITEs to TABLEA would have identical effects:

```
A  WRITE TABLEA AREA1
   WRITE TABLEA 100/50
   WRITE TABLEA AREA1/50
```

The field defined by data-area must hold the key immediately followed by the descriptor.

Note: Examples of the use of table-handling verbs appear at the end of this chapter.

6.8.4 LOOKUP Verb and *TABLEs

LOOKUP is a condition verb that searches a table by key, and indicates whether an entry with a particular key already exists. Its format is:

```
LOOKUP   tablename   data-area
```

'tablename' is the name of a table previously defined on a *TABLE directive.

'data-area' defines a field whose contents are used to locate an entry during the LOOKUP operation. It may be any valid character Field Definition, the name of a field already defined in the *DICTIONARY, or, at a BREAK Decision Point only, an FSC preceded by an asterisk (*FSC). If the length of data-area is different from the key length of the table entry, the search is performed using the shorter key.

Unlike the other table-handling verbs, LOOKUP is a condition verb, and must, therefore, have a condition entry supplied. A Y (yes) reply means that the key is present (a true LOOKUP); an N (no) reply means that the key is not present (a false LOOKUP).

Note: A LOOKUP searching for a value X'FF' always returns TRUE, because of the presence of the 'end-of-table' entry.

The action of the LOOKUP verb is not affected by the actions of any previous table-handling verbs.

Examples of the use of table-handling verbs appear at the end of this chapter.

6.8.5 SELECT Verb and *TABLEs

SELECT is an action verb that retrieves an entry from a table. Its format is:

```
SELECT   tablename   field
```

'tablename' is the name of a table previously defined on a *TABLE directive.

'field' is a signed binary field of length four, or a numeric constant. The value of the field SELECTs that entry from the table. The first entry is entry number zero.

It is important not SELECT an entry whose value is greater than the total number of entries in the table.

The action of the SELECT verb is not affected by the actions of any previous table-handling verbs.

Note: Examples of the use of table-handling verbs appear at the end of this chapter.

6.8.6 DELETE Verb and *TABLEs

DELETE is an action verb that deletes an entry from a table. The entries are deleted only for the duration of the current run. The format of DELETE is:

```
DELETE   tablename
```

'tablename' is the name of a table previously defined on a *TABLE directive.

If a DELETE follows a READ, SELECT or true LOOKUP, then that current record is deleted.

If a DELETE follows a WRITE, the record following the one written is deleted.

If a DELETE follows a false LOOKUP, the record with the next highest key value is deleted. An error results, however, if the key value is greater than that of the highest key in the table.

Note: Examples of the use of table-handling verbs appear at the end of this chapter.

6.8.7 CLOSE Verb and *TABLEs

CLOSE is an action verb that deletes all the entries in a table for the duration of the current run. This means that any subsequent access of a

CLOSEd *TABLE re-opens it as though it had been specified without any entries. The format of CLOSE is:

```
CLOSE   tablename
```

'tablename' is the name of a table previously defined on a *TABLE directive.

6.8.8 'Key-only' *TABLEs

*TABLEs may be set up with no descriptors; only Ks appear on the entry format statement. Such tables are normally used for validation, the presence or absence of the key being the criterion for acceptance or rejection. The most common form of access is through the LOOKUP verb, although it is possible to WRITE to such a table.

6.8.9 Examples of *TABLE Access

The *TABLE to which the following examples refer is:

```
*TABLE COLOURS
K  DDDDD
1  RED
2  BLUE
4  GREEN
6  PUCE
```

6.8.9.1 Example 1: Successive READs

```
*DETAB ONE
A  READ  COLOURS  100/5
   READ  COLOURS  100/5
   READ  COLOURS  100/5
```

Here, the first descriptor returned would be 'RED', the second would be 'BLUE', and the third would be 'GREEN'. If two further READs were performed, the 'end-of-table' descriptor (X'FFFFFFFFFF') would be returned. Any subsequent READ would return to the beginning of the table and progress normally.

6.8.9.2 Example 2: READ Following 'true' LOOKUP

```
*DETAB TWO
I  200/1      MV    '4'
C  LOOKUP  COLOURS  200/1     Y  N
A  READ    COLOURS  100/5     X  .
```

Here, the descriptor returned would be 'GREEN', since the key '4' was successfully sought by the LOOKUP.

6.8.9.3 Example 3: READ Following 'false' LOOKUP

```
*DETAB THREE
I  200/1      MV    '5'
C  LOOKUP  COLOURS  200/1     Y  N
A  READ    COLOURS  100/5     .  X
```

Here, the descriptor returned would be ’PUCE’. Since the key ’5’ was unsuccessfully sought by the LOOKUP, the descriptor corresponding to the next higher key would be returned.

6.8.9.4 Example 4: READ Following SELECT

```
*DETAB FOUR
I  200:4      MV    3
   SELECT  COLOURS  200:4
   READ    COLOURS  100/5
```

Here, since numbering begins at zero, ’3’ refers to the fourth item in the table. Consequently, the descriptor returned would be ’PUCE’.

6.8.9.5 Example 5: WRITE Following ’true’ LOOKUP

```
*DETAB FIVE
I  200/1      MV    '4'
   201/5      MV    'PINK'
C  LOOKUP  COLOURS  200/1     Y  N
A  WRITE   COLOURS  200/6     X  .
```

Here, the descriptor ’GREEN’ would be overwritten with ’PINK’, since the key ’4’ was successfully sought by the LOOKUP.

6.8.9.6 Example 6: WRITE Following ’false’ LOOKUP

```
*DETAB SIX
I  200/1      MV    '5'
   201/5      MV    'GREY'
C  LOOKUP  COLOURS  200/1     Y  N
A  WRITE   COLOURS  200/6     .  X
```

Here, the descriptor written would be ’GREY’. Since the key ’5’ was unsuccessfully sought by the LOOKUP, it would be inserted, with ’GREY’ as its corresponding descriptor. The insertion would be made between ’GREEN’ and ’PUCE’.

6.8.9.7 Example 7: WRITE Following SELECT

```
*DETAB SEVEN
I  200:4      MV    3
   SELECT  COLOURS  200:4
   200/6      MV    '5GREY'
   WRITE   COLOURS  200/6
```

Here, since numbering begins at zero, ’3’ refers to the fourth item in the table. Consequently, the descriptor ’GREY’, and the key ’5’ would be written between the current third and fourth items.

6.8.9.8 Example 8: DELETE Following ’true’ LOOKUP

```
*DETAB EIGHT
I  200/1      MV    '2'
C  LOOKUP  COLOURS  200/1   Y  N
A  DELETE  COLOURS          X  .
```

Here, the key '2' was successfully sought by the LOOKUP, so the key, and the descriptor 'BLUE', would be DELETEd.

6.8.9.9 Example 9: DELETE Following 'false' LOOKUP

```
*DETAB NINE
I  200/1      MV    '5'
C  LOOKUP  COLOURS  200/1   Y  N
A  DELETE  COLOURS          .  X
```

Here, since the key '5' was unsuccessfully sought by the LOOKUP, the key and descriptor with the next higher value (that is, '6 PUCE') would be DELETEd.

6.8.9.10 Example 10: DELETE Following READ

```
*DETAB TEN
I  READ    COLOURS  200/6
   DELETE  COLOURS
```

Here, the first access was a READ, so the item accessed would be the first key and descriptor in the table ('1 RED'); this is therefore the item DELETEd.

If, in this example, the access had been performed by SELECT, the item DELETEd would have been the item SELECTed.

6.9 Worked Example

This example is a variation on the program shown in *Worked Example on page 39 of Chapter 3*.

A report is produced showing Area, Account Number and Balance for customers whose details are held on the customer master file. Customers with Account Numbers in the range 9000-9799 are to be excluded. The salesman's name is to be printed. A grand total Balance is to be printed at the end. Records are to be printed in Account Number sequence.

```
*PROGRAM EXAMPLE
*FILE  CUSTFILE RL=332  NAME=custfile.txt
*DICTIONARY
AREA      = 2/2      [ AREA
ACCNO     = 4/4      [ ACCOUNT NUMBER
BALANCE   = 324:4    [ BALANCE
SALESMAN  = -14/14   [ SALESMAN
*INLIST
A AREA
B ACCNO
C SALESMAN
2 BALANCE
*TABLE SALES
KK DDDDDDDDDDDDDD
21 F. JOHNSON
22 D. ALL
28 J. SMITH
37 M. WILLIAMS
92 M. ROBERTS
```

```
*HEAD L CH1,2
DD/MM/YY     CUSTOMER BALANCE LISTING     PAGE PPPP



  AREA       SALESMAN        ACCOUNT     BALANCE
*OUT L 1,0
   AA        CCCCCCCCCCCCCC  BBBB       22222.22
*OUT F 3,0
  'FINAL TOTAL'                       2222222.22
*DETAB RECORD
C ACCNO   GE '9000'        Y  ELSE   [ IGNORE
  ACCNO   LE '9799'        Y  .      [  UNWANTED
A IGNORE                   X  .      [  RECORDS
  CALL    GETSALES         .  X      [ FIND SALESMAN
*DETAB GETSALES
C LOOKUP  SALES AREA       Y  N      [ SEARCH TABLE
A READ    SALES SALESMAN   X  .      [ EXTRACT ENTRY
  SALESMAN S               .  X      [ NO ENTRY IN TABLE
*SORT ACCNO
*GO
```

A sample of the output appears below:

```
20/10/00     CUSTOMER BALANCE LISTING     PAGE    1

  AREA       SALESMAN        ACCOUNT     BALANCE

   21        F. JOHNSON      1144        1511.20
   21        F. JOHNSON      1313         151.50
   22        D. ALL          3211          16.84
   22        D. ALL          3270         128.62
   21        F. JOHNSON      8643         109.22
   21        F. JOHNSON      9827         210.80


   FINAL TOTAL                          64303.77
```

Chapter 7
Reserved Words, Restricted Names

7.1 Reserved Words

A number of fields have been allocated to help the programmer access special data fields used by the Fixed Logic. The names given to these fields are reserved and must not be defined in the `*DICTIONARY`. This chapter describes these 'Reserved Words'. The Reserved Words used by FILETAB are:

Name	Definition	Use
`BINDATE`	`:4`	Holds the number of days elapsed since December 31 1899
`BREAK`	`/1`	Holds the level of the current control break
`COMPDATE`	`/8`	Holds the date when the program was compiled
`COMPTIME`	`/8`	Holds the time when the program was compiled
`COMPVERS`	`/4`	Holds the version number of the compiler when the program was compiled
`COUNT`	`:4`	Holds the value 1. Normally associated with a totalling FSC to indicate the number of records selected in each control group
`DATE`	`/8`	Holds the current date in the format DD/MM/YY
`DATEYYYY`	`/10`	Holds today's date in DD/MM/YYYY format
`DD`	`/2`	Holds the day from the current date
`FF`	`:4`	Holds the length in bytes of the fixed portion of variable length records from the main file
`LEVEL`	`/1`	Holds the control level being processed. It contains a character between `L` and the current control break, or `F`

Name	Definition	Use
LINES	:4	Holds the decrementing line count. This is either the number of lines available for printing on the current page or, if no lines are available for printing on the current page, a number less than or equal to zero. *Note: The line count takes into account any paper control required after the printing of the last line of a print block. If the last line printed is not the last line of a print block, a throw of one line after printing is included in the count*
LOCALTIME	:36	LOCALTIME is taken from the system clock and adjusted according to time zone. Each word of LOCALTIME has the following values: LOCALTIME:4 seconds after minute (0-59) LOCALTIME+4:4 minutes after hour (0-59) LOCALTIME+8:4 hours since midnight (0-23) LOCALTIME+12:4 day of month (1-31) LOCALTIME+16:4 month (0-11, 0=January) LOCALTIME+20:4 year (since - 1900) LOCALTIME+24:4 day/week (0-6, 0=Sunday) LOCALTIME+28:4 day/year (0-365, 0=Jan 1) LOCALTIME+32:4 +ve if daylight saving time is current, equal to zero if not or -ve if daylight saving status is unknown. LOCALTIME can be converted into various formats by using EXECUTEd routines(see *EXECUTE Verb on page 1* of *Chapter 10*)
MARK	/4	Holds the version number of the compiler
MM	/2	Holds the month from the current date
NOPRINT	:4	When set to one, the current print line is not printed. Defaults to zero
PAGE	:4	Holds the current page number
PP-3	/1	Holds the control level of the line being printed
PP-2	/1	Holds H if the line being printed is a heading, O if it is an output line, or T if it is a title
PP-1	:1	Holds the number of the line within the print group
PP0	:1	Holds a FILETAB printer control character. 0 No line feed +n n line feeds before line -n thrown to line n (for example, -1 = CH0)

Name	Definition	Use
PP1-160	/160	Holds the line to be printed
PROGRAM	/8	Holds the name of the program being executed
RECLEN	:4	Holds the length of the current *FILE record, but also has special uses, such as deferring reading of the main file
RECOUNT	:4	Holds the number of records read from the main file
REMAINS	:8	Holds the remainder from the most recent division operation. It may be positive or negative, depending on the sign of the dividend, and whether the rounding was up or down. In all cases, the dividend equals the quotient multiplied by the divisor, plus REMAINS
RPN	/1	Holds the control level at which the page number is reset to 1
SUBCOUNT	:4	Holds the number of sub-records processed from the current main file record
TIME	/8	Holds the time at the start of the current program, in the form hh.mm.ss
VV	:4	Holds the length in bytes of the variable portions of variable length records from the main file.
YY	/2	Holds the year from the current date
YYYY	/4	Holds the year number as four digits at the current date
ZREPLY	:4	Holds the reply after a READ/WRITE to an Indexed-Sequential file which has an alternate index. ZREPLY is also used to hold the return value of a routine called using the EXECUTE verb (see *EXECUTE Verb on page 1* of *Chapter 10*)

These Reserved Words may be used in the *INLIST (in order to print them) or in decision tables at the relevant Decision Point.

7.1.1 Use of BREAK, LEVEL and PP-3

LEVEL contains a control character representing the current control level. At the start of a control break cycle, it is set to the level of the current control break (that is, it is set equal to the character held in BREAK). During the *HEAD printing sequence, LEVEL descends through the intermediate levels to L. After the *HEAD sequence, the level of the next control break is ascertained and stored in BREAK; LEVEL is then set to L, and ascends, during the *OUT sequence, through the intermediate

levels, to the level of the new control break. LEVEL may be accessed at the BREAK and PRINT Decision Points.

Note: At the PRINT Decision Point, PP-3 contains a copy of the value held in LEVEL.

7.1.2 Use of FF, VV and SUBCOUNT

The details which follow also appear, together with a fuller description and an example of use, in *Use of FF, VV and SUBCOUNT on page 3* of *Chapter 6*.

If the Main File contains variable length records in sub-record format, the *FILE directive should contain the keywords FF and VV. The Reserved Words FF and VV may also be included in decision tables, to vary the defined values.

The length in bytes of the fixed portion of a record is defined initially by the FF keyword. This may be altered by a decision table at the RECORD level of processing, which moves a value into the Reserved Word FF.

The length in bytes of the variable portion of a record is defined initially by the VV keyword. Again, this may be amended by a decision table at the RECORD level of processing.

FF and VV are initialised with the *FILE values only at the start of the run. After alteration it is therefore a good idea to test and ensure that they continue to hold the correct values.

When a record on the Main File has a length equal to FF, SUBCOUNT is set to -1.

When a record on the Main File has a length smaller than FF, SUBCOUNT is set to -2.

When a record has a variable portion not exactly divisible by VV, SUBCOUNT is set to -3.

7.1.3 Use of COUNT

The binary field COUNT contains 1; if it is defined in the *INLIST as a totalling field, it can be used to give a count of the number of records in a control group. For example:

```
*FILE  TRANSDY  RL=64  NAME=transdy.txt
*DICTIONARY
VALUE    = 28:4  [ VALUE IN PENCE
AREA     = 2/2   [ AREA
ACCNO    = 4/4   [ ACCOUNT NUMBER
*INLIST
N AREA
M ACCNO
2 VALUE
3 COUNT
*OUT M 1,1
'TOTAL VALUE FOR ACCOUNT' MMMM  £22222.22   3333 'RECORDS'
*OUT N 2,1
```

```
'TOTAL VALUE FOR AREA'NN       £222222.22   3333 'RECORDS'
*GO
```

7.1.4 Use of RECLEN

RECLEN is a signed binary field which contains the length (in bytes) of the latest record read from the main file. Its two other functions are to control the passage of records through the Fixed Logic, and to simulate end-of-main-file. For full details of the use of RECLEN, see *Use of RECLEN at the RECORD Decision Point on page 11 of Chapter 4.*

7.1.5 Use of LINES

If LINES is set to -1 at a BREAK Decision Point, it causes a new page to be taken before the *OUT print block for that control level is printed. It also prints *HEAD L CHn, if present.

Note: LINES may not be used to influence spacing between print lines. This function is performed by PP0 (see below).

7.1.6 Use of PP0

PP0 contains the format effector for the current print line. This may be altered at the PRINT Decision Point to influence pagination (see *Printer Control Characters (Format Effectors) on page 45 of Chapter 4*).

7.1.7 Access to Reserved Words

The following table indicates the Decision Point(s) at which each Reserved Word may be accessed, and the kind(s) of access allowed at these points:

Name	Decision Point	Access
BINDATE	any	READ
BREAK	BREAK	READ
COMPDATE	any	READ
COMPTIME	any	READ
COMPVERS	any	READ
COUNT	any	READ
DATE	any	READ
DATEYYYY	any	READ
DD	any	READ
FF	RECORD, ENDFILE	READ, WRITE
LEVEL	BREAK, PRINT	READ
LINES	BREAK	READ, WRITE
LOCALTIME	any	READ
MARK	any	READ
MM	any	READ
NOPRINT	START, PRINT	READ, WRITE
PAGE	PRINT	READ
PP-3	PRINT	READ
PP-2	PRINT	READ
PP-2	PRINT	READ
PP-1	PRINT	READ
PP0	PRINT	READ, WRITE
PP1-160	PRINT	READ, WRITE
PROGRAM	any	READ
RECLEN	START, RECORD, ENDFILE	READ, WRITE
RECOUNT	RECORD, ENDFILE	READ
REMAINS	any	READ
RPN	any	READ
SUBCOUNT	RECORD, ENDFILE	READ
TIME	any	READ
VV	RECORD, ENDFILE	READ, WRITE
YY	any	READ
YYYY	any	READ
ZREPLY	any	READ

7.2 Restricted Names

7.2.1 Restricted Decision Table Names

In general, names beginning with the letter `Z` are given to any decision tables integral to the FILETAB software. It is best therefore, to avoid using such names for decision tables.

The following decision table names have special meanings:

`BREAK`	`BREAK` Decision tables
`BREAKH`	`BREAKH` Decision tables
`ENDFILE`	`ENDFILE` Decision table
`PRINT`	`PRINT` Decision table
`RECORD`	`RECORD` Decision table
`START`	`START` Decision table

Chapter 8
Field Definition and Pointers

A 'Field Definition' describes the field's position, type and length. Field Definitions are normally associated with 'Fieldnames' (by which they may be referred to later in the program) by entries in the `*DICTIONARY`. A general description of Field Definitions appears in **DICTIONARY Definition on page 12 of Chapter 2.*

A 'Pointer' is a data item which is used to perform indirect addressing and string-handling in FILETAB programs.

Pointers contain address variables when they are referenced within decision tables. The specification and use of Pointers are described in this chapter, and several examples are included in *Chapter 11.*

8.1 Alternative Field Definition Formats

Data items are described in Field Definitions, which specify the position, type and length of fields. The basic format is:

```
start-position   type   length
```

`start-position` may be given as:

- An unsigned integer defining a byte displacement in the Record Area
- A negatively-signed integer defining a byte position in the Work Area
- A print position preceded by the letters `PP`
- A card column number preceded by the letters `CC`

`type` may be given as:

- / for character fields
- : for signed binary fields
- ; for unsigned binary fields

When `CC` or `PP` is used, type must be character (/).

`length` defines the length of the field in bytes.

There must be no embedded spaces in Field Definitions.

Besides this standard format of Field Definition, the following alternative formats are available:

(1) `start-position - end-position`

This format is available only for character fields. `start-position` must commence `CC` or `PP`; `end-position` denotes the card column or print position in which the field terminates. This format may be used both in `*DICTIONARY` definitions and in decision tables. For example:

```
*DICTIONARY
 CARD  =  CC1-80
 PRINT =  PP1-120

*DETAB RECORD
A  CC10-13  MV 'DEPT'
*
*DETAB PRINT
A  PP40-43  MV 'AREA'
```

(2)
```
@@   type   length
     type   length
```

No start-position is specified here, and it is implied to be the next free byte in the Record or Work Area, depending on whether the previous field was in the Record or Work Area. For the first Field Definition, the start-position is zero. This form of definition is allowed only in `*DICTIONARY` entries.

This method of defining fields is particularly useful when record formats are liable to be changed, since the absence of specified start-positions means that additional fields can be inserted without alteration to existing `*DICTIONARY` entries.

(3) `name + offset type length`

This format allows definition of fields relative to the start of a previously-defined field called 'name'. When the offset is zero and the type is character (/) or binary (: or ;), the format may be:

(4) `name type length`

In `*DICTIONARY` definitions, this may be further simplified to:

(5) `name`

This format implies that the type and length are the same as those of the previously-defined field. The following example illustrates these points.

```
*DICTIONARY
 FIELD  =  4/4
 FLD1   =  FIELD       [ 4/4
 FLD2   =  FIELD:4     [ 4:4
 FLD3   =  FIELD+4/4   [ 8/4
```

```
*DETAB EXAMPLE
I FIELD:4   MV   FLD3/4
  FIELD+4/2  S
```

(6) `(indirect-start-position)   type   length`

The indirect-start-position enables the field to be addressed at run-time through the use of Pointers, and must be a four byte binary field, preferably word-aligned. It must be specified within round brackets. `type` can be character (`/`) or a binary (`:`).

For example:

```
(POINT1):4
(4:4)/4
```

(`indirect-start-position`) may also be specified as `(indirect-start-position)+offset`. In this case, `offset` is added to the value of (`indirect-start-position`) each time the field is accessed.

(7) `(indirect-start-position)  type  ±  length  ±`

The rules for using indirect-start-position, type and length are as stated in 4 above. The automatic increment/decrement (±) is used to update the indirect-start-position by the length before, after, or both before and after, the operation is performed. For example:

```
(POINT1):+4-
(POINT2)/6+
(POINT3)/-8
```

(8) `(indirect-start-position)type  (indirect-length)`

This is identical to format 4, apart from (`indirect-length`). This is used when the length of a field is not known at compile-time. (`indirect-length`) must be a Field Definition for a four-byte binary field, preferably word-aligned, which contains the length of the field, in bytes, at run-time. For example:

```
(POINT1):(LEN1)
```

(9) `(indirect-start-position)type ± (indirect-length) ±`

This is identical to format 6, apart from automatic increment/ decrement, which is as described for format 5.

(10) `start-position   type   (indirect-length)`

This format is used when the length of the field is not known at compile-time. (`indirect-length`) must be the Field Definition of a four-byte binary field, preferably word-aligned, that contains the length of the field in bytes at run-time.

For example:

```
FIELD/(L1)
```

(11) `start-position`

Neither a type nor a length is specified here. It is implied that a one-byte character field is required.

Further information on formats 4 to 9 is given in *Introduction to Pointers on page 5* and *Example of an Application using Pointers on page 10*.

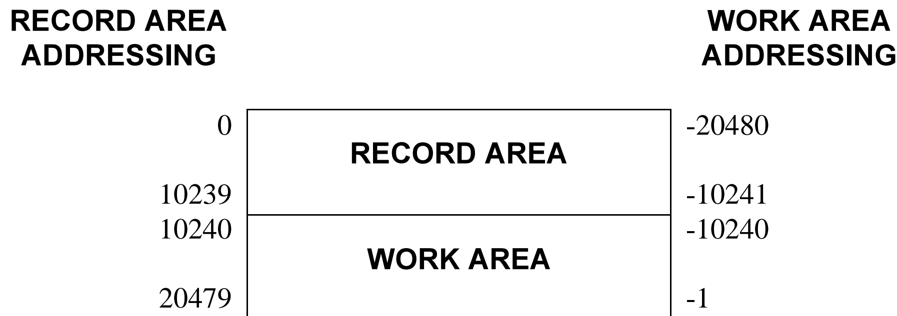
8.2 Restrictions on Field Start-positions and Lengths

The size of the Record Area is 10240 bytes. However, Record Area addresses may exceed this, up to an absolute maximum of 20480 bytes. The fields being referenced would be in the Work Area. This allows the whole of the work space (the Record and Work Areas) to be considered as one large Record Area, and defined as such.

The size of the Work Area is 10240 bytes. However, Work Area addresses may exceed this, up to an absolute maximum of 20480 bytes. The fields being referenced would be in the Record Area. The field definitions 0/10 and -20480/10 address the same storage area.The sum of a start-position and length must not exceed zero. For example:

```
*DICTIONARY
WORKA = -4/4      is valid
WORKB = -4/5      is invalid
WORKC = 20470/20
```

The following diagram illustrates the addresses of these areas:



Field lengths are subject to the following restrictions:

Field type	Restriction
Character	No practical limit, except for the combined size of the Record and Work Areas
Binary	Signed binary (defined by `:`) Unsigned binary (defined by `;`) *Note: There is no practical field size limit, but fields longer than the machine's word length are used inefficiently*

8.3 Introduction to Pointers

The fields discussed in previous chapters have usually been defined in fixed positions within the Record and Work Areas:

```
FRED = 100/4, WORK = -200:4  etc
```

where every reference to `FRED` would access character `100` onwards, and every reference to `WORK` character `-200` onwards. This technique may be unsuitable, however, when the field in question is not always to be found in a fixed position. For instance, if the record to be read consists of a fixed portion followed by several sub-records of differing lengths, or if an array is to be accessed, the address of a required field may not be known at compile-time.

To simplify matters, FILETAB uses a method whereby fixed fields are used to point to the variable address. These fixed fields are called 'Pointers'. The following example indicates the value of using Pointers.

A file contains two fields. The first (a `:4` field) contains a Region Code, which may be zero, one, two or three; the second (a `:8` field) contains a value. The file is to be read, and a total is to be produced for each region. The two methods by which the necessary totalling might be performed are illustrated below.

In this first version, `*DETAB ONE` does not use Pointers; it is written in limited entry format, because an extended entry version might be misleadingly concise - FILETAB always converts into limited entry internally.

```
*DICTIONARY
 REGION =   0:4         [FROM INPUT
 VALUE  =   4:8         [FROM INPUT
 TOT0   = 100:8
 TOT1   = 108:8
 TOT2   = 116:8
 TOT3   = 124:8
*DETAB ONE
C  REGION  =  0         Y  N  N  N
   REGION  =  1         .  Y  N  N
   REGION  =  2         .  .  Y  N
   REGION  =  3         .  .  .  Y
A  TOT0    +  VALUE     X  .  .  .
   TOT1    +  VALUE     .  X  .  .
   TOT2    +  VALUE     .  .  X  .
   TOT3    +  VALUE     .  .  .  X
```

In the following alternative version, `*DETAB TWO` does use Pointers. It is an action-only table, so no testing is necessary.

```
*DICTIONARY
 REGION   =   0:4  [FROM INPUT
 VALUE    =   4:8  [FROM INPUT
 PFIELD   = 200:4  [TO BE USED AS A POINTER
 TOT0     = 100:8
 TOT1     = 108:8
 TOT2     = 116:8
 TOT3     = 124:8
```

```
*DETAB TWO
A  PFIELD       MV  A'TOT0
   REGION       *   8
   PFIELD       +   REGION
   (PFIELD):8   +   VALUE
```

Here, the address of TOT0 is loaded into a Pointer. A 'TOT0 *is* an Address Constant; these are discussed in *Initialising Pointers on page 6.* Since the Pointer is held as a character offset, it follows that, for it to point to TOT2, it would have to be incremented by 16. To calculate the appropriate offset for the above example, the value held in REGION is multiplied by eight, which is the length of each field. To update the Pointer, the offset resulting from this calculation is added to it. To indicate to FILETAB that the contents of a field are to be used as a Pointer, its fieldname or *DICTIONARY definition is enclosed within round brackets. The final action in *DETAB TWO adds the contents of VALUE to the field pointed at by PFIELD.

8.4 Pointer Definitions

Any four byte binary field (that is, :4) can be used as a Pointer, and its Field Definition is the same regardless of whether it is to be used as a Pointer or as a pure binary data field. Pointers are more efficient, however, if they are word-aligned. A field is word-aligned if its start position is exactly divisible by four.

All the following examples might be used as Pointers:

```
*DICTIONARY
P1    = -4:4
PTR   = 1288:4
ADDR  = -16:4
P2    = 300:4
```

8.5 Initialising Pointers

To ensure that a Pointer always contains a valid reference to a FILETAB data area, it must be initialised only by use of an 'Address Constant'. Thereafter, arithmetic operations may be used to modify the data address to which it refers.

An Address Constant is a constant which enables FILETAB to generate an absolute store address when the program is executed. The general form of an Address Constant is:

```
A'operand
```

The operand may be any Field Definition (except that of a Pointer Field; see *Pointer Fields on page 9*) which is acceptable in a *DICTIONARY; a start position in the Record or Work Area; or a Field Specifying Character immediately preceded by an asterisk (see *Chapter 3*). The use of absolute start positions should be avoided, as this programming style may result in programs being excessively difficult to maintain.

The following example shows initialisation of Pointers:

```
*DICTIONARY
NAME = 8/35
WORK = -20/20
P1   = -24:4
P2   = -28:4
P3   = 132:4
P4   = -32:4
*INLIST
C NAME
*DETAB ANY
A P1 MV A'NAME  [ P1 POINTS TO START OF NAME
  P2 MV A'WORK  [ P2 POINTS TO START OF WORK
  P3 MV A'*C    [ P3 POINTS TO START OF NAME IN *INLIST AREA
  P4 MV A'100   [ P4 POINTS TO CHAR 100 OF RECORD AREA
```

Note: `P3` is initialised to contain the address of the field `NAME` as it is stored in the `*INLIST` Area. This instruction is valid only when obeyed at the `BREAK` Decision Point.

Once a Pointer has been initialised by use of an Address Constant, it can be modified by arithmetic operations. Alternatively, the Address Constant can be used after arithmetic operations, as in the following example. As software developers are not aware of the absolute location of storage areas within their programs, arithmetic operations only generate displacements from an Address Constant. For further details on the modification of Pointers, see *Updating Pointers on page 8 and Pointer Fields on page 9.*

In the next example, a record has twelve four-byte binary fields, starting at position 80, containing quantities ordered in each month of a year. The first field relates to January, the second February and so on. The coding below shows one way to compute the address of a particular field.

```
*DICTIONARY
P1     = -4:4       [ POINTER
TOTALS = 80/48      [ ACCUMULATORS
MONTH  = -124/2     [ MONTH FROM TRANSACTION RECORD
*DETAB FINDTOT
A P1 MV MONTH       [ MONTH NUMBER
  P1 *  4           [ MULTIPLY BY LENGTH OF TOTAL
  P1 -  4           [ STEP BACK ONE TOTAL
  P1 +  A'TOTALS    [ MODIFY BY START ADDRESS
```

So far, only fixed length Pointers have been considered, where the length is expressed as a constant following the field type symbol. When the length is unknown at compile-time, however, ’indirect-length’ Pointer operations must be used. An indirect length is held in a four-byte binary field, preferably situated on a word boundary. The format is:

```
start-position   type   (indirect-length)
```

`start-position` is as described in *Alternative Field Definition Formats on page 1.*

`type` is `:` ; or /

’`indirect-length`’ is a `*DICTIONARY` name or a valid field definition that, at run-time, contains a valid length for the operation.

For example, if the field FLD was defined in the *DICTIONARY as having a length of 20 characters, the action

```
100:4 MV 20
```

would move the value 20 into 100:4. This field could then be used as an indirect length. However, if the length of FLD were changed subsequently, the program would need amendment. To avoid this, a 'length constant' could be used to provide the length at run-time. The general format of a length constant is:

```
L'operand
```

The operand may be any field, other than a Pointer field (see *Pointer Fields on page 9*), that has been specified in the *DICTIONARY, or a Field Specifying Character preceded by an asterisk (see *Chapter 3*).

The statement:

```
100:4 MV L'FLD
```

would load the binary number 20 into 100:4. This binary word could then be used as an indirect-length Pointer:

```
(POINT)/(100:4) MV ADDRESS
```

Alternatively, 100:4 could be defined as LEN in a *DICTIONARY and that name could be used both when the length is loaded and in the MOVE statement:

```
LEN MV L'FLD
(POINT)/(LEN) MV ADDRESS
```

The length, however obtained, must always be a positive integer, and be within the limits for the operation.

8.6 Updating Pointers

As Pointers themselves can be considered to be pure binary data fields, their contents can be changed at any time by performing arithmetic operations on them. The following example searches backwards through the field NAMEADDR until the character * is found.

```
*DICTIONARY
NAMEADDR = 20/144           [ VARIABLE N/A
P1       = -4:4             [ POINTER
*
*DETAB SEARCH
A P1     MV A'NAMEADDR      [ START OF AREA
  P1     +  143             [ END OF AREA
  GOTO   SEARCH2            [ START SEARCHING
*
*DETAB SEARCH2
C (P1)/1 EQ '*'   Y N       [ ASTERISK?
A P1     -  1     . X       [ STEP BACK
  REPEAT          . X       [ CHECK AGAIN
  GOTO   FOUND    X .       [ EXIT
```

This routine, however makes the assumption that an asterisk ALWAYS occurs within the field NAMEADDR. Since FILETAB has no way of estimating the end of the search area, the routine continues until an asterisk is found. The address of this asterisk could be outside the data-space, which would almost certainly result in a logic error. Checks should always be performed to ensure that Pointers remain within specified limits. The example shown above could be better written as follows:

```
*DETAB SEARCH
I P1     MV   A'NAMEADDR+143
C P1     LT   A'NAMEADDR       Y N N    [ END OF SEARCH?
  (P1)/1 EQ   '*'              . Y N    [ ASTERISK?
A P1     -    1                . . X    [ STEP BACK
  REPEAT                       . . X    [ CHECK AGAIN
  GOTO   FOUND                 . X .    [ EXIT
  GOTO   NOTFOUND              X . .    [ NO ASTERISK
```

Several changes have been made. Firstly, SEARCH and SEARCH2 have been combined into a single decision table by the use of an initial action. Secondly, an offset has been included in the line that set P1 to its initial value. Lastly, and most importantly, a check has been incorporated, to ensure that the Pointer operation is terminated at the beginning of the NAMEADDR buffer.

8.7 Pointer Fields

Fields referenced via the contents of a Pointer are referred to as 'Pointer Fields'. Pointer Fields may be defined in a *DICTIONARY or may be explicitly defined in decision tables. The general format is:

```
(Pointer) type  ±  length  ±
```

Pointer is any four byte signed binary field, preferably word-aligned. When the contents of a Pointer are to be used as an indirect address, the Pointer must be enclosed by round brackets.

type is the field type and may be any type.

'length' is the length (in bytes) of the area being addressed indirectly. The length itself may not be known until run-time. In this case, a *DICTIONARY name or a valid field definition, enclosed by round brackets, may be used. When the length is calculated, it should be placed in this field.

A + or − sign before the length indicates that the Pointer is to be incremented or decremented by the direct or indirect length, before the Pointer is used to access the Pointer Field; a + or − sign after the length indicates that the Pointer is to be incremented or decremented after use.

8.7.1 Examples

(P1)/4	Address of field is in P1
(P2)/+3	Add 3 to P2, then use P2 as the field address.
(P3)/2-	Address of field is in P3. After use, subtract 2 from P3
(P4)/-1+	Subtract 1 from P4 and then use P4 as the data address. After use, add 1 to P4 (that is, restore the original address)
(P5)/(L1)	Address of field is in P5, and its length is in field L1
(P6)/+(L2)	Add the contents of L2 to P6, and then use P6 as the field address, using the contents of L2 as the length
(P7)/(L3)-	Address of field is in P7, and length in field L3. After use, subtract the contents of L3 from P7
(P8)/+(L4)-	Add the contents of L4 to P8, and then use P8 as the field address, using the contents of L4 as the length. After use, subtract the contents of L4 from P8 (that is, restore the original address)

Any of the above field definitions could be associated with a name in the *DICTIONARY, which could then be used in the decision table statement.

8.8 Operations using Pointer Fields

Either or both of the operands in a decision table detail may be specified as Pointer Fields with or without automatic increment/decrement. Some examples of valid Pointer operations appear below.

```
(P1)/4+   MV '****'    [ MOVE ASTERISKS THEN UPDATE P1
-16:4     +  (P3):4    [ ADD IN INDIRECT BINARY WORD
(PTR1)/1+ MV (P2)/1+   [ INDIRECT MOVE WITH AUTO-INCREMENT
```

Pointer Fields cannot be used with one-of-a-set tests.

When Pointer Fields are used in decision tables, it is possible to add an offset to, or subtract an offset from, the Pointer value. For example:

```
(P1)+10/4+  MV  '****'
```

Here, four asterisks would be moved into the field starting ten bytes beyond the byte indicated by the Pointer. The Pointer would be incremented by four after the operation.

Note: The offset does not affect the value stored in the Pointer.

```
(P1)-10/4   MV  '****'
```

Here, four asterisks would be moved into the field starting ten bytes before the byte indicated by the Pointer.

Note: The offset does not affect the value stored in the Pointer.

8.9 Example of an Application using Pointers

One method of reducing storage space is to minimise the number of characters used to store a name and address. In a typical application, one hundred and eighty bytes may be required for a name and address, being

six lines of thirty bytes each on printing. In order to compact a name and address, line delimiters are used so that the address:

```
A.N. OTHER
25 SMITH STREET
NEWTOWN
SALOP
```

is stored as:

```
A.N. OTHER*25 SMITH STREET*NEWTOWN*SALOP***
```

Note: For ease and speed of processing, all six line terminators are used.

The coding below assumes that a compacted name and address is stored in the field `NAMEADDR`. The name and address is to be expanded into a fixed-length area, `LABEL`, in the form of six lines, each of thirty characters.

```
*DICTIONARY
NAMEADDR = 20/144       [ NAME AND ADDRESS
LABEL    = -180/180     [ BUILD AREA
CTR      = -181P1       [ TERMINATOR COUNTER
P1       = -188:4       [ POINTER
P2       = -192:4       [ POINTER
P3       = -196:4       [ POINTER
*
*DETAB EXPAND
I LABEL    S                  [ CLEAR RECEIVING AREA
  P1       MV A'NAMEADDR      [ POINTER TO NAME/ADDRESS
  CTR      Z                        [ TERMINATOR COUNTER
  P2       MV A'LABEL               [ START OF CURRENT LINE
  P3       MV P2                    [ RECEIVING FIELD
C (P1)/1   EQ '*'       Y Y ELSE    [ SEE IF TERMINATOR
  CTR      EQ 5         Y N .       [ END OF NAME/ADDRESS?
A (P3)/1+  MV (P1)/1+   . . X       [ MOVE ONE BYTE
  P1       +  1         . X .       [ STEP OVER TERMINATOR
  P2       +  30        . X .       [ STEP TO NEXT LINE
  P3       MV P2        . X .       [ RESET RECEIVING ADDRESS
  CTR      +  1         . X .       [ TERMINATOR COUNT
  REPEAT                . X X
```

8.10 Worked Example

This section shows a further variation of the example which first appeared in *Worked Example on page 29 of Chapter 2*, and demonstrates some of the techniques introduced in this chapter.

A report is produced from the customer master file (`CUSTFILE`) showing Area, Account Number, Balance, Name and Address. Only customers whose address contains the string '`WALES`' are to appear on the report. Selected records are printed in Account Number sequence.

```
*PROGRAM EXAMPLE
*FILE  CUSTFILE RL=332  NAME=custfile.txt
*DICTIONARY
AREA     = 2/2         [ AREA
ACCNO    = 4/4         [ ACCOUNT NUMBER
NAMADDR  = 8/140       [ NAME AND ADDRESS
```

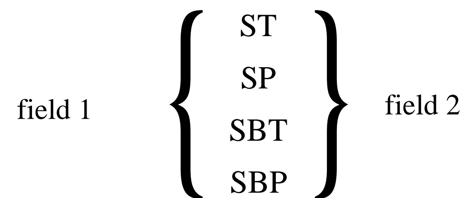
```
ADDR     = 44/104     [ ADDRESS
BALANCE  = 324:4      [ BALANCE
*
P1       = -4:4       [ POINTER TO NEXT CHARACTER
P2       = -8:4       [ POINTER TO ADDRESS END
*INLIST
A AREA
B ACCNO
C NAMADDR
2 BALANCE
*HEAD L CH1,2,55
DD/MM/YY         WELSH CUSTOMER LISTING            PAGE PPPP

AREA  A/C   NAME AND ADDRESS                         BALANCE
*OUT L 1,1
 AA   BBBB  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC   £22222.22

            CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

            CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

            CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

            CCCCCCCC
*DETAB RECORD
I P1       MV A'ADDR                 [ START OF ADDRESS
  P2       MV P1                     [ END OF
  P2       +  100                    [  SEARCH AREA
C P1       LT P2           Y Y       [ STILL WITHIN ADDR...
  (P1)/5   EQ 'WALES'      Y N       [ WALES...
A P1       +  1            . X       [ STEP ON ONE CHARACTER
  REPEAT                   . X       [ KEEP ON LOOKING
*SORT  ACCNO
*GO
```

A sample of the output appears below:

```
20/08/00         WELSH CUSTOMER LISTING            PAGE    1
AREA  A/C   NAME AND ADDRESS                         BALANCE

 37   2100  JOHNSONS (DEMOLITION) LTD.                $30.00
            0-24 NEATH ROAD
            CARDIFF
            SOUTH WALES

 21   8643  B AUTOS LTD.                             £109.22
            SWALESDALE VIEW
            YORK
```

Note: The second record has been selected because the string '`WALES`' occurs within '`SWALESDALE`'.

8.11 Pointer Skip Operators



'field1' is a Pointer Field (that is, a four byte binary field containing an address attribute). Note that, although it is a Pointer Field, it is not enclosed in round brackets.

'field2' is either a character Field Definition or a character constant.

The action of ST (Skip To) is to increment field1 until the Pointer value it contains points to a character that is one of the set of characters held in, or defined by, field2.

The action of SP (Skip Past) is similar, except that, on completion of the action, field1 will be pointing to the first character found that is not one of the set held in, or defined by, field2. Note that field2 itself may be a Pointer, if required.

The action of SBT (Skip Back To) is to decrement field1 until the Point value it contains points to a character that is one of the set of characters held in, or defined by field2.

The action of SBP (Skip Back Past) is similar to SBT, except that on completion of the action, field1 will be pointing to the first character that is not one of the set held in, or defined by field2.

It is the user's responsibility to ensure that a character that will terminate the ST or SP action does actually appear in the area beyond the position initially pointed at by the contents of field1. If there is no such character, results are indeterminate, and the program will fail.

Where SP is employed, the Pointer will usually be pointing initially at one of the set of characters held in, or defined by, field2, when the SP action is obeyed. If this is not so, the Pointer value will not be altered.

Similarly, the ST action will not alter the value of the Pointer, if it is already pointing at one of the set of characters held in, or defined by, field2.

```
*PRO SKIPS
*DIC
PNAME  = 0/25                        [NAME ON INPUT FILE
PNAME2 = 100/25                      [CONVERTED NAME
QT     = 125/1                       [CHARACTER TO CONTAIN A QUOTE
MOD1   = 128:4                       [POINTER FOR ORIGINAL FIELD
MOD2   = 132:4                       [POINTER FOR CONVERTED FIELD
LEN1   = 136:4                       [LENGTH OF INPUT NAME
```

```
LEN2   = 140:4                  [LENGTH OF CONVERTED NAME
.
.
.
*DETAB REMOVEQT                 [REMOVE FIRST QUOTE FROM NAME
I PNAME2 MV PNAME               [MOVE NAME TO CONVERTED NAME
  MOD1   MV A'PNAME             [POINTER TO START OF NAME
  MOD2   MV A'PNAME2            [ START OF CONVERTED NAME
  QT;1   MV 39                  [BINARY VALUE OF QUOTE
*
C PNAME  CT  QT/1               Y N [NAME CONTAINS QUOTE?
A MOD1   ST  QT/1               X . [SKIP TO QUOTE
  MOD1   SP  QT/1               X . [SKIP PAST QUOTE
  MOD2   ST  QT/1               X . [SKIP TO QUOTE
  LEN1   MV A'PNAME+25          X . [LENGTH OF REST ..
  LEN1   -  MOD1                X . [ .. OF NAME
  LEN2   MV LEN1                X . [ADD 1 TO ..
  LEN2   +   1                  X . [ SECOND LENGTH
  (MOD2)/(LEN2) MV (MOD1)/(LEN1) X .
*         MOVE REST OF NAME - A SPACE ADDED AT END
.
.
*DETAB LASTQT                   [REMOVE LAST QUOTE FROM NAME
I PNAME2 MV PNAME               [MOVE NAME TO CONVERTED NAME
  MOD1   MV A'PNAME+24          [POINTER TO END OF NAME
  MOD2   MV A'PNAME2+24         [ END OF CONVERTED NAME
  QT;1   MV 39                  [BINARY VALUE OF SINGLE QUOTE
*
C PNAME  CT  QT/1               Y N [NAME CONTAINS QUOTE?
A MOD1   SBT QT/1               X . [SKIP BACK TO QUOTE
  MOD2   SBT QT/1               X . [SKIP BACK TO QUOTE
  MOD1   SP  QT/1               X . [SKIP PAST QUOTE
  LEN1   MV A'PNAME+25          X . [LENGTH OF REST ..
  LEN1   -  MOD1                X . [ .. OF NAME
  LEN2   MV LEN1                X . [ADD 1 TO ..
  LEN2   +   1                  X . [ SECOND LENGTH
  (MOD2)/(LEN2) MV (MOD1)/(LEN1) X .
*         MOVE REST OF NAME - A SPACE ADDED AT END
.
*GO
```

Chapter 9
Techniques

This chapter deals in some detail with programming style, and with the techniques that lead to more efficient use of FILETAB.

9.1 Calculation and Field Manipulation

9.1.1 Multiplying a Binary Field by a Power of 2

When a binary field is to be multiplied by a power of 2, the recommended method is the use of a shift. For example:

```
-4:4 SL 3
```

is faster than:

```
-4:4 * 8
```

It is also possible to use shifts to perform division by a power of 2. This may only be done, however, when the binary field is known to contain a positive number, since these shift actions are logical shifts.

9.1.2 Arithmetic on Record Area Fields

A Record Area field should not be moved into the Work Area solely for the performance of arithmetic operations, unless it is essential for the field in the Record Area to remain as it was on the input file. For example, coding `A` is better than coding `B`:

```
A                    B

*DICTIONARY          *DICTIONARY
FIELDA = 108:8       FIELDA  = 108:8
                     WFIELDA = -8:8
*                    *
*INLIST              *INLIST
 1  FIELDA            1  WFIELDA
*                    *
*DETAB RECORD        *DETAB RECORD
A  FIELDA  *  25     A WFIELDA  MV  FIELDA
   FIELDA  /  100      WFIELDA  *   25
                       WFIELDA  /   100
```

9.1.3 Field Types

When performing calculation or field manipulation, FILETAB allows different field types on either side of the operator. When the fields on both sides of the operator are of similar type, FILETAB can perform the operation directly. For example:

```
60/4  +  70/4
-4:4  +  -8:4
```

In the above examples, each operation can be performed without any conversion of the formats of the fields. It should be noted that binary to binary operations are of the same order of efficiency.

When field types are dissimilar, as in the following example, one or both of the fields must be converted by FILETAB, prior to the operation:

```
4:4  +  8/4
4:4  +  '123'
```

It should be noted here that FILETAB cannot perform arithmetic functions directly on character fields, so that, when character fields (or literals) are used in this context, a conversion is always implied.

The following piece of calculation exemplifies the kinds of conversion performed by FILETAB. Of the different example codings, coding `A` is better than coding `B`.

```
A                 B

-4:4  MV  4/6     4/6  *  15
-4:4  *    15     4/6  / 100
-4:4  /   100
 4/6  MV -4:4
```

At first sight, coding `B` would appear to be the more efficient, but, within the machine, arithmetic can be performed only on binary fields. These two versions of this calculation would generate the following events:

A	B
- Convert character to binary	- Convert character to binary
- Multiply binary	- Multiply binary
- Divide binary	- Convert binary to character
- Convert binary to character	- Convert character to binary
	- Divide binary
	- Convert binary to character

Coding B demonstrates that it is particularly inefficient to perform more than one arithmetic operation on a single character field.

9.1.4 Operations on Character Strings

The time taken for operations to be performed on character strings depends on the length of the strings themselves. Consequently, it is advisable to keep the length of character strings down to that which is actually required.

9.1.5 Calculation to a Fixed Number of Decimal Places

FILETAB only handles integers. However, when calculations are being performed, it is often desirable to generate and preserve a fixed number of decimal places. This may be done by scaling the fields used in the calculation by a power of 10, and using the decimal point (.) editing character for output. For example, to convert inches to centimetres, one would normally multiply by 2.54. If a result to two decimal places was required, the following technique could be employed.

```
*DETAB CONVERT
*
* CONVERT INCHES TO CENTIMETRES
I CM MV INCH
  CM  *  254
*INLIST
 1  CM
*OUT L
   111.11
```

After the conversion, the field `CM` is 100 times too large. However, the inclusion of the decimal point editing symbol means that, on output, the right-most two digits are printed as decimals, thus giving the correct value.

If the output were required to the nearest centimetre, the following technique could have been adopted:

```
*OPTION DR
*DETAB CONVERT
*
* CONVERT INCHES TO CENTIMETRES
I CM MV INCH
  CM  *  254
  CM  /  100
```

9.2 Decision Tables

This section describes some of the ways in which decision tables may be most efficiently used, and indicates some of the areas in which care should be taken.

9.2.1 Structure of Programs

It is important to avoid thinking in flowchart terms at the detail level. Instead, a modular approach should be adopted, and design should be directed towards the efficient utilisation of decision tables.

Subject to the notes about the number of rules in decision tables (see *Decision Tables on page 9*), it is usually worthwhile increasing the number of rules in a decision table, if one or more conditions or actions can be removed as a result.

9.2.2 Conditions

It is not uncommon for the same condition test to appear more than once in a program. Where this occurs, it usually means that it is possible to combine two or more decision tables into one.

Whenever possible, conditions should be ordered in such a way as to enable FILETAB's relevance testing mechanism to avoid the evaluation of irrelevant conditions.

In general, those conditions which eliminate most rules should appear first. For example:

```
*DETAB A
C  FLD  >  2     N  Y  Y  Y
   FLD  >  4     .  N  Y  Y
   FLD  >  6     .  .  N  Y

*DETAB B
C  FLD  >  6     N  N  N  Y
   FLD  >  4     N  N  Y  .
   FLD  >  2     N  Y  .  .
```

One of these two decision tables is more efficient than the other, depending on the most common value of `FLD`. This is because conditions with irrelevance entries (.) in all satisfied rules are not evaluated. Thus, if `FLD` were less than three, in `*DETAB A` only the first condition would be evaluated, while all three conditions would be evaluated in `*DETAB B`.

The `IF` verb should be used carefully, since relevance testing may prevent the `IF`ed decision table being entered:

```
C   A  =  B     Y  Y  N
    IF  DETC    Y  N  .
```

Here, `*DETAB DETC` is entered only if '`A`' equals '`B`'. If `*DETAB DETC` is required to perform initialisation or other tasks irrespective of the outcome of the first condition, this may not give the desired results. The same applies to any condition that may change the value of FILETAB fields, for example `LOOKUP` and Pointer operations, where auto-increment or decrement is specified.

9.2.3 ELSE Rule

Where the `ELSE` rule, either implicit or explicit, is likely to be obeyed quite frequently in a decision table, the conditions should be ordered to allow the `ELSE` rule to be entered as quickly as possible; this avoids unnecessary evaluation of conditions.

9.2.4 Actions

When the same decision table is to be entered more than once, `REPEAT` should be used, rather than `CALL` or `GOTO`. `REPEAT` is faster than `GOTO self`, and returns to the first condition in the decision table, thus avoiding any initial actions. When it is necessary to initialise some work fields, this should be done in the first decision table of the program, outside the

program's logic cycle, rather than in some later decision table which is entered many times.

Actions in the most frequently obeyed rule in a decision table should, wherever possible, be grouped together at the beginning of the action section, especially when the last action in the group is a `REPEAT` or a `GOTO`. This minimises the number of action masks evaluated by FILETAB.

Actions with identical action entries should, when possible, be grouped together. Only one action mask is generated for adjacent actions with identical entries. For example:

```
A  FIELDA  MV  FIELDB   X  .  X
   FIELDC  +   FIELDD   X  .  X
   CALL DETA            X  .  X
   GOTO DETB            X  X  .
```

In the above example, the first three actions are compounded into one long action, because of their similar entry portions. This reduces the number of tests on the action mask.

9.2.5 Extended Entry Format

It should be borne in mind that the extended entry format is purely a shorthand way of expressing decision table conditions and/or actions. FILETAB does not optimise the code generated. The following examples illustrate this point.

9.2.5.1 Conditions in Extended Entry

The condition line:

```
C   FIELD = ?    2  4  6  8  10
```

generates the following limited entry lines:

```
C   FIELD = 2    Y  .  .  .  .
    FIELD = 4    .  Y  .  .  .
    FIELD = 6    .  .  Y  .  .
    FIELD = 8    .  .  .  Y  .
    FIELD = 10   .  .  .  .  Y
```

Even if `FIELD` is equal to 2, FILETAB has to evaluate all five conditions. If the decision table had been written in limited entry format, efficiency could have been improved thus:

```
C   FIELD = 2    Y  N  N  N  N
    FIELD = 4    .  Y  N  N  N
    FIELD = 6    .  .  Y  N  N
    FIELD = 8    .  .  .  Y  N
    FIELD = 10   .  .  .  .  Y
```

In this case, if `FIELD` is equal to 2, only the first condition is evaluated, because of FILETAB's relevance testing. Because of this, FILETAB evaluates a condition only if the result can affect the rule or rules satisfied in the table. In the example above, rules 2 to 5 are known to be false, because `FIELD` is equal to 2. Since none of the following conditions can

affect the true rule (rule 1), they are not evaluated. Maximum efficiency is therefore obtained when the most frequently obeyed rule is specified first.

The following example explains why FILETAB converts extended entries into `Y` and `.` entries, rather than into `Y`, `N` and `.` entries. Consider the following condition:

```
C  FIELD  >  ?  2  4  6  8  10
```

If FILETAB were to insert `N` entries to the right of the `Y`, incorrect results would be obtained. This is indicated by the resultant limited entry version:

```
C  FIELD  >  2  Y  N  N  N  N
   FIELD  >  4  .  Y  N  N  N
   FIELD  >  6  .  .  Y  N  N
   FIELD  >  8  .  .  .  Y  N
   FIELD  >  10 .  .  .  .  Y
```

Here, if `FIELD` were equal to 7, rule 1 would be satisfied. The intention, however, was that rules 1, 2 and 3 should all be satisfied.

9.2.6 Identical Entries in Extended Entry Decision Tables

Where two or more identical entries are specified in an extended entry condition or action, the order of them can affect the efficiency of the table. This is because FILETAB combines extended entries which are lexically identical and adjacent into a single condition or action, but does not combine those which are lexically identical but are not adjacent. The following examples should clarify this point.

The following action line:

```
A  FIELD  MV  ?       2  4  2  6  2
```

generates the following limited entry lines:

```
A  FIELD  MV  2       X  .  .  .  .
   FIELD  MV  4       .  X  .  .  .
   FIELD  MV  2       .  .  X  .  .
   FIELD  MV  6       .  .  .  X  .
   FIELD  MV  2       .  .  .  .  X
```

Here, rules 1, 3 and 5 are not combined.

If the rules in the decision table were rearranged to allow the action line to be written thus:

```
A  FIELD  MV  ?       2  2  2  4  6
```

the following limited entry lines would have been generated:

```
A  FIELD  MV  2       X  X  X  .  .
   FIELD  MV  4       .  .  .  X  .
   FIELD  MV  6       .  .  .  .  X
```

Here, rules 1, 2 and 3 are combined, because they are both lexically identical and adjacent.

The following extended entry action line:

```
A  FIELD  MV  ?      2   02   2   4   6
```

generates the following limited entry lines:

```
A  FIELD  MV  2       X    .    .    .    .
   FIELD  MV  02      .    X    .    .    .
   FIELD  MV  2       .    .    X    .    .
   FIELD  MV  4       .    .    .    X    .
   FIELD  MV 6        .    .    .    .    X
```

Here, rules 1, 2 and 3 are not combined. This is because, although logically they produce the same result, `2` and `02` are not lexically identical.

9.2.7 Decision Points

Tests designed to `IGNORE` unnecessary records should be performed as soon as possible at the `RECORD` level. If there is a `*SORT`, and records cannot be `IGNORE`d at the `RECORD` level, it is better to `IGNORE` them at `*DETAB BREAK` than at `*DETAB PRINT`.

Because it is performed for every line of printing, processing at `*DETAB PRINT` can be inefficient. Consequently, it should be avoided or minimised, wherever possible.

9.3 Pointers

During Pointer operations, the automatic increment/decrement facility (*see Chapter 8.8*) should be employed whenever possible, since it is faster than separate arithmetic operations on the Pointer itself.

A useful point to remember is that field comparisons are terminated when the left-hand field is exhausted. This facilitates the use of such techniques as the following:

```
C  CC1-2  =  (P)/8+   Y  N
```

Here, the first two characters of the field pointed at by the Pointer `P` are tested for equality with `CC1-2`. After the test, the Pointer is incremented by eight. In this way, the first two characters of a set of contiguous `/8` fields could be tested quite simply.

When a Pointer test is used in a recursive decision table, an additional counter need not always be used to end the recursion. An Address Constant may instead be assigned to the end of the field, and the Pointer value may be tested against it. When the value of the Pointer is greater than that of the Address Constant, recursion is ended. For example:

```
I  POINTER       MV  A'CC1
   COUNTER       MV  10
C  COUNTER        >  0         Y  Y  N
   (POINTER)/3+  =  'ABC'     N  Y  .
A  COUNTER        -  1         X  .  .
   REPEAT                      X  .  .
   GOTO FOUND                  .  X  .
```

is less efficient than:

```
I   POINTER       MV  A'CC1
C   POINTER        >  A'CC30   N  N  Y
    (POINTER)/3+  =   'ABC'    N  Y  .
A   REPEAT                     X  .  .
    GOTO FOUND                 .  X  .
```

Special care must be taken when automatic increment/decrement is used in condition lines (as in the above example), because the effect of relevance testing may mean that certain conditions are not evaluated, and that, consequently, the increments/decrements which form part of such conditions are not performed.

One factor essential to the efficient use of Pointers is alignment. This is discussed in *Alignment on page 8.*

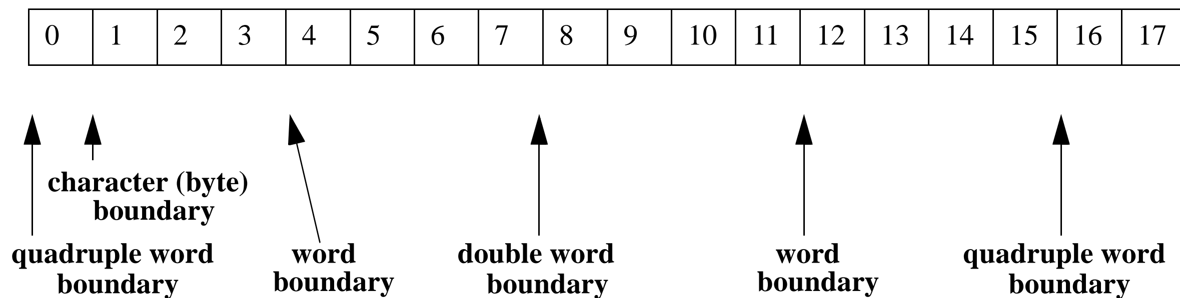
9.4 Alignment

9.4.1 Binary Fields

Efficiency considerations for binary fields are dependent upon the architecture of the machine in use. There are two factors to consider: word size and alignment. The word sizes and alignments of a selection of machines are below:

Processor	Makes	Word Size	Alignment	Pointer Size
Intel	ICL TeamServer I series UNISYS, Linux (for Intel), Windows	4 bytes	1 byte	4 bytes
Sparc	ICL TeamServer S series SUN	4 bytes	4 bytes	4 bytes

Binary fields can be any number of bytes long and placed at any byte address (for example 7:11). For them to be processed most efficiently however, they should be the same length as the machine's word size (usually `:4`) and start at an address, offset into the Record Area, which is a multiple of the machines's alignment (that is, a word boundary, see example below).



The Work Area is mapped in a similar way. Address `-16`, for example, is on a quadruple word boundary.

Maximum efficiency is obtained if the following rules are observed:

(1) Wherever possible, binary fields should be defined as same length as the machine's word size. For example:

```
:4, :8
```

(2) `:4` fields should start on single, double or quadruple word boundaries; that is, with start-positions exactly divisible by `4`.

(3) `:8` fields should start on double or quadruple word boundaries; that is, with start-positions exactly divisible by `8`.

(4) Where fields are not a multiple of four bytes long, or where they do not start on a word boundary, the number of word boundaries crossed should be minimised. It is particularly inefficient to cross a quadruple word boundary. Thus, for instance, `15:2` is less efficient than `1:2`.

9.4.2 Character Fields

For character fields, it is most efficient to use the shortest field possible.

9.4.3 Pointer Fields

Pointer fields must be the same length as the machine's pointer size (usually the word size) and for maximum efficiency should start at an address which is a multiple of the machine's alignment (that is, a word boundary). Similar rules apply to fields that hold indirect lengths.

9.5 Internal Format

9.5.1 Numeric Constants

Numeric Constants may be held in one of three internal formats. The format used is determined by the value of the Numeric Constant, as follows:

- Integers in the range +63 to -64 are held in the most efficient format
- Integers in the range +64 to +131071, and the range -65 to -131072, are held in a marginally less efficient format
- Integers outside these ranges are held in the least efficient format

9.5.2 Decision Tables

Decision table rule masks are held as Numeric Constants, and their efficiency depends on the number of rules in the table, as follows:

- Decision tables with six rules or less are processed most efficiently
- Decision tables with 7 to 17 rules are processed slightly less efficiently
- Decision tables with more than 17 rules are processed least efficiently

9.5.3 Grouping of Fields

Most machines have special hardware to allow faster access to frequently-used areas of store. To make best use of this feature, it is advisable to group fields in one area of working storage. For example, it may be more efficient to use these three fields:

```
-4:4    -8:4    -12:4
```

than to use these three:

```
-4:4    -3000:4    -6000:4
```

9.6 *INLIST

The `*INLIST` should include no more FSCs than are necessary for printing and the use of fields at `*DETAB BREAK`. In particular, redundant control fields should not be specified in the `*INLIST`.

It is preferable for `*INLIST` Control and Transfer fields to be as short as possible. One way to minimise their length is to use coded data. For example, if the Control field represents a Sales Region, regions could be given numbers (`Region 1 = NORTH WEST`, and so on). This would mean that the size of data files and sort work files would be reduced, and so would the amount of data manipulated by the Fixed Logic.

When a report was being produced, the coded data could be decoded before printing by the use of the `*LOOKUP BREAK` facility (see **LOOKUP BREAK x on page 19 of Chapter 4*).

The use of dummy totalling fields (see *Dummy Totalling Fields on page 4 of Chapter 4*) is recommended in situations where calculations are performed using other totalling fields (for example to calculate a percentage), since these fields are not included in sort records.

9.7 Headings and Print Lines

In `*HEAD` and `*OUT` lines, the use of editing symbols and floating symbols, and the use of re-cycling and truncation of fields, should be minimised whenever possible.

Output lines containing no editing symbols at all, take a fast path, with a consequent improvement in efficiency.

Truncation of fields is inefficient, so fields in print lines should always be made large enough to hold the maximum possible value. For example, moving the value `132157` to a field defined as `66666` is inefficient; the field should be changed to `666666`.

Signing a field through the use of `*OPTION PSxx`, or `*OPTION NSxx`, is less efficient than the inclusion of the appropriate floating symbol.

Chapter 10
External Routines

10.1 EXECUTE Verb

It may be appropriate or convenient to incorporate within FILETAB programs, routines written in other programming languages (such as ‘C’). The mechanism within FILETAB used to facilitate this is the `EXECUTE` verb. It has the following possible formats:

```
EXECUTE name
EXECUTE name(parameter1)
EXECUTE name(parameter1, parameter2, ...)
```

Parameters are passed to the called routine as pointers (if within FILETAB they are character fields, hexadecimal or character literals) or integers (if within FILETAB they are signed/unsigned binary fields or numeric literals). Any value returned by the called routine is placed in the reserved word `ZREPLY`.

10.1.1 Example

Date and time conversions can be achieved by executing various C library functions to convert date and time formats. For example, the function `strftime` can be used to convert a field in the same format as `LOCALTIME` (see) into various formats:

```
EXECUTE strftime(CD_Field, Length, Format, Time_Field)
```

Parameter	Explanation
CD_Field	The destination character field for the formatted date
Length	The length of the field

Parameter	Explanation
Format	A character literal or character field, specifying the format to be produced. If a character field is used the format characters should be followed by a terminating character value 0 (X'00'). The field or literal can contain characters to be copied across, and format directives introduced by%. Some of the format directives are: %b abbreviated 3 character month name %d day of month as 2 digits %j day of year as 3 digits %m month as 2 digits %y year without the century as 2 digits %Y year with the century as 4 digits
Time_Field	A thirty six byte field in the same format as LOCALTIME described in *Reserved Words on page 1 of Chapter 7*.

The following section of code:

```
*DIC
   date    -16/16
                  [15 chars. for date+1 for terminating char.
   timefld -52/36       [copy of LOCALTIME
   years   timefld+20:4 [years within timefld
   format  -58/5        [for format YYYYDDDD
*DET one
I  date MV LOCALTIME+12:4
   date/12 MV 'Day of Month'
   DISPLAY date
   EXECUTE strftime(date, 16, '%a %d/%b/%Y', LOCALTIME
*                   date in form: Mon 23/Apr/2001
   DISPLAY date
   timefld MV LOCALTIME
   years +1
   format MV '%Y%j'
  format+4/1 MV X'00' [could do EXECUTE strcpy(format, '%Y%j')
   EXECUTE strftime(date, 16, format, timefld)
   DISPLAY date/7
```

would display:

```
Day of Month  23
MON 23/Apr/2001
2001113
```

Chapter 11 Examples

11.1 Simple Listing

This example demonstrates the use of FILETAB purely for listing data held on a sequential file.

As no tabulation is involved, only transfer fields appear in the `*INLIST`, and the report is produced in the original file sequence (that is, account number sequence).

```
*PROGRAM TEST1
*FILE CUSTFILE RL=332  NAME=custfile.txt   [ CUSTOMER FILE
*OPTION PE
*DICTIONARY
 CACCNO   = 4/4               [ CUSTOMER ACCOUNT NUMBER
 ACNAME   = 8/36              [ ACCOUNT NAME
*INLIST
 A CACCNO
 B ACNAME
*HEAD L CH1,2,55
     ACCOUNT NO.     CUSTOMER NAME
*OUT L 1,0
     AAAA            BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
*STOP
```

A sample of the output from this report is shown below.

```
     ACCOUNT NO.     CUSTOMER NAME


     1144            SMITH'S TIMBER LTD
     1313            R.S. WILDE JOINERY
     1364            QUINLAN MANUFACTURING CO. LTD.
     1397            M.V. HUMPAGE (NORTH) LTD.
     1487            OLDHAM PLASTICS
     2100            JOHNSONS (DEMOLITION) LTD.
     2180            DYER GARNETT & PARTNERS
     2238            M.V. HUMPAGE (SOUTH) LTD.
     2441            MERCANTILE CARRIERS LTD.
     4243            CORBETT CONSTRUCTION COMPANY
     7173            STEP-RITE SHOE CORP.
     8431            SOMERSET CEREAL SUPPLIERS
     8643            JB AUTOS LTD.
     9187            EASTERN ROADWAYS LTD.
     9212            OFFICE CLEANERS (SW) LTD.
     9213            NEWTOWN PAINT SUPPLIES
     9221            RICHARDSON & GREEN
     9298            WATERHOUSE (UK) LTD.
     9768            MATTHEW AND SON
     9827            RSJ SUPPLIES
     9832            B & W (ADVERTISING) LTD.
```

11.2 Listing with Final Totals

FILETAB is frequently used for the computation of totals in order to present a simple report.

In this example, the customer master file (`CUSTFILE`) is read, and a line is printed for each record. Account number and balance are shown, and balances are totalled at the end. The report is produced in account number sequence (that is, file sequence).

Note: In the `*INLIST`, the balance is assigned a totalling FSC, which invokes FILETAB's tabulation process.

```
*PROGRAM TEST2
*OPTION PE,ZS
*FILE  CUSTFILE  RL=332  NAME=custfile.txt
*DICTIONARY
 CACCNO   = 4/4       [ ACCOUNT NUMBER
 CBALANCE = 324:4     [ BALANCE
*INLIST
 A CACCNO
 1 CBALANCE
*HEAD L CH1,2,55
       ACCOUNT

       NUMBER            BALANCE
*OUT L 1,0
       AAAA            111111.11
*OUT F 3,0
  'FINAL TOTAL'       1111111.11
*STOP
```

A sample of the output from this report follows.

Note: Since the balance field has been assigned a totalling FSC in the `*INLIST`, zeros are suppressed when the balance is printed.

If you refer to the source statements for this report, remember that in order to generate the two line heading, four source statements have been provided. (See *Reserved Words on page 1 of Chapter 7* for further details). Remember also, when totalling, that extra digits may have to be catered for on printing. Though experience tells us that no single balance exceeds eight digits, we have made provision for nine digits to be printed in the final total. If any total is too large to fit its picture image, asterisks are output on printing, though the correct value is carried forward in the *INLIST Area to the higher level (final) total.

```
       ACCOUNT
       NUMBER            BALANCE

       1144              1511.20
       1313               151.50
       1364              2009.50
       1397                 0.91
       1487                 9.50
       2100                30.00
       2180                 1.25
       2238               980.22
       2441               590.10
       4243               591.50
```

```
        7173              492.56
        8431                7.51
        8643              109.22
        9187               12.50
        9212                1.25
        9213             1211.50
        9221               90.50
        9298                8.00
        9768              200.30
        9827              210.80
        9832                0.90

    FINAL TOTAL          8220.72
```

11.3 Listing with Control Break Totals

This program produces a report from the transaction file (TRANSDY), showing the total value of transactions for each account number. Records on this file are in random sequence, but several transactions are likely to appear for a single customer.

A grand total is to appear at the end. The report is produced double-spaced.

```
*PROGRAM TEST3
*OPTION PE,NSDR,ZS
*FILE TRANSDY RL=64 NAME=transdy.txt  [ TRANSACTION FILE
*DICTIONARY
 TACCNO   = 4/4                     [ TRANSDY ACCOUNT NUMBER
 TVALUE   = 28:4                    [ TRANSACTION VALUE
*INLIST
 M TACCNO
 2 TVALUE
*HEAD L CH1,3,57
DD/MM/YY     ANALYSIS OF ORDER VALUES


ACCOUNT NO.           TOTAL VALUE
*OUT M 0,2
  MMMM              £22222222.22
*OUT F 1,0
                    '------------'


'TOTAL VALUES'     £2222222222.22
*SORT
*STOP
```

In this program, the report is produced in account number sequence, because of the *SORT directive. The specification of TACCNO as a control field in the *INLIST enables us to print a line for each account number (*OUT M), and the specification of TVALUE as a totalling field causes FILETAB to tabulate this data.

*Note: To embed a blank line in the heading, we have had to generate a blank line in the *HEAD detail statements. This, of course, requires two blank source statements, making six statements in all for the*

heading details. The current date is incorporated in the heading by the string `DD/MM/YY`*.(See Chapter 2.1.7 for further details).*

In the `*OPTION` detail statement, we have included the option `NSDR`; this ensures that any negative totals are indicated on the report, though, even if we omitted this, the sign would be taken into account during totalling. A floating `£` sign appears in the `*OUT` detail statement.

A sample of the output from this program appears below.

```
20/08/00     ANALYSIS OF ORDER VALUES

ACCOUNT NO.           TOTAL VALUE

  1144                     £14.75DR

  1313                   £3789.88

  2100                   £9141.25

  2441                   £1980.98

  9221                   £3754.07

  9298                   £9217.88

  9768                   £1982.87

                       ------------
 TOTAL VALUES           £29852.18
```

As an alternative to `*OPTION NS`, a floating minus sign (`-`) could have been specified in the detail lines associated with `*OUT M` and `*OUT F`. For example:

```
*OUT M 0,2
  MMMM              £-22222222.22
```

The first record would then have been printed as follows:

```
  1144                    £-14.75
```

Note: In the above example, a totals-only `*SORT` *(see Chapter 4.4) would be performed by default, since there is no* `*LOOKUP BREAK L`*,* `*DETAB BREAK L` *or* `*OUT L` *directive present.*

11.4 Listing with Selection and Flagging

In this example, the customer master file (`CUSTFILE`) is read, and details of credit code and outstanding balance are printed. A customer's details appear on the report only if his credit code is less than `06`. It is believed that a credit code of `00` has been allocated wrongly to some customers, and this is to be checked. No sorting is required, as the report is to be in account number sequence.

```
*PROGRAM TEST4
*OPTION PE,ZS
*FILE CUSTFILE  RL=332  NAME=custfile.txt
*DICTIONARY
```

```
 CACCNO   = 4/4      [ CUSTOMER ACCOUNT NUMBER
 CBALANCE = 324:4    [ OUTSTANDING BALANCE
 CREDCODE = 328/2    [ CREDIT CODE
*
 ERRFLAG  = -5/5     [ WILL HOLD 'ERROR' OR SPACES
*INLIST
 A CACCNO
 B CREDCODE
 C ERRFLAG
 2 CBALANCE
*HEAD L CH1,2,56
DD/MM/YY               CREDIT  ANALYSIS           PAGE  PPPP

 CUSTOMER              CREDIT  CODE                  BALANCE
*OUT L 1,0
 AAAA                  BB      CCCCC               £22222.22
*DETAB RECORD
C CREDCODE LT '06'    Y      Y     [ SELECT IF CRCODE LT 06
  CREDCODE EQ '00'    Y      N     [ SEE IF CRCODE INVALID
A ERRFLAG  MV ?    'ERROR'  ' '    [ SET ERROR FLAG
*STOP
```

Here, we must make provision for the printing of a data item which is not present on the Main File. Depending on the value of CREDCODE, the word ’ERROR’ may appear. Of course, the nonappearance of a field is usually achieved by outputting spaces in its place. In the *DICTIONARY, we have reserved a five byte character field named ERRFLAG. As this field is in the Work Area, its contents are under our exclusive control, and it is not even initialised by FILETAB.

Record selection is performed at the RECORD Decision Point, where we have specified a single decision table. The first condition rejects all records where CREDCODE is not less than 06. This is as a result of the implicit ELSE (IGNORE) rule generated for all FILETAB decision tables.

Note: A credit code of ' ' is regarded as less than 06. The second condition identifies records containing the invalid credit code 00. When this invalid credit code is present, ERRFLAG is set to the value 'ERROR' for subsequent printing. We must ensure, however, that whenever the credit code is valid the word 'ERROR' does not appear, so we set ERRFLAG to spaces.

In the *INLIST, CBALANCE appears as a totalling field, even though no control fields have been declared, and no *OUT F is present. Although the data is essentially transfer, we are thus able to use the full editing functions of totalling fields.

Note: In the heading, the current date has been included via DD/MM/YY, and automatic page numbering has been requested via PPPP.

A sample of the output follows:

```
27/08/00              CREDIT  ANALYSIS           PAGE     1
 CUSTOMER             CREDIT  CODE                  BALANCE

 1313                 01                            £151.50
 1397                 01                              £0.91
 2100                 00       ERROR                 £30.00
 2180                 05                              £1.25
```

```
4243                03                          £591.50
7173                05                          £492.56
8431                05                            £7.51
9187                02                           £12.50
9212                05                            £1.25
9221                01                           £90.50
9298                02                            £8.00
9768                03                          £200.30
9827                05                          £210.80
9832                                              £0.90
```

11.5 Processing at the RECORD Decision Point

This example demonstrates the incorporation of simple processing at the `RECORD` Decision Point. For each record on the file `TRANSDY`, a physical quantity is computed by multiplying together the quantity and the quantity code.

Only records for area 21 are printed, and records which are printed appear in reference number sequence.

```
*PROGRAM TEST5
*FILE TRANSDY RL=64  NAME=transdy.txt
*OPTION PE,ZS
*DICTIONARY
TRANS-REC                = 0/64
T-AREA                   = TRANS-REC+/2
T-REF                    = TRANS-REC+14/6
T-QCODE                  = TRANS-REC+55/3
T-QTY                    = TRANS-REC+60:4
*DICTIONARY
 WK-QTY    = -4:4   [ WORK AREA FOR PHYSICAL QUANTITY THAT
                    [ IS, TQCODE * T-QTY
*INLIST
 A T-REF
 B T-QCODE
 1 T-QTY
 2 WK-QTY
*HEAD L CH1,2,55
DD/MM/YY           AREA 21 TRANSACTIONS            PAGE PPPP


 REFERENCE       QUANTITY          QUANTITY        PHYSICAL

 NUMBER                            CODE            QUANTITY
*OUT L 1,0
 AAAAAA           111111           BBB             22222222
*DETAB RECORD
C T-AREA EQ '21'           Y         [ IGNORE ALL EXCEPT
A GOTO CALCQTY             X         [  AREA 21
*
*DETAB CALCQTY
C T-QCODE EQ '?'      SNG   PR    HDZ   DOZ   GRS
A WK-QTY  MV T-QTY    X     X     X     X     X
  WK-QTY  *  ?        .     2     6     12    144
*
*SORT T-REF
*STOP
```

Note: The programmer has explicitly defined work fields by means of a `*DICTIONARY` directive. The quantity code is also not held as a numeric value, and must therefore be interpreted before multiplication with the quantity.

In the decision table `RECORD`, records with an area other than 21 are ignored, as a result of the implicit `ELSE` rule.

A sample of the report follows.

```
03/08/00             AREA 21 TRANSACTIONS         PAGE    1
REFERENCE       QUANTITY          QUANTITY        PHYSICAL
NUMBER                            CODE            QUANTITY

DC2138                17          SNG                   17
DC2141                 3          GRS                  432
MS0108               200          SNG                  200
MS0110                15          HDZ                   90
MS0124                 6          DOZ                   72
RC2010                30          HDZ                  180
```

11.6 Processing at the BREAK Decision Point

This example shows a simple application which requires processing to be performed at a `BREAK` Decision Point. A report is produced in which account number and balance are to appear for each customer. For each area, a total balance and an average balance are to be printed.

In this program, only areas 21 and 37 are represented.

```
*PROGRAM TEST6
*FILE  CUSTFILE RL=332  NAME=custfile.txt
*OPTION PE,ZS
*DICTIONARY
 AREA     = 2/2       [ AREA
 ACNO     = 4/4       [ ACCOUNT NUMBER
 BALANCE  = 324:4     [ BALANCE
*INLIST
 M AREA
 A ACNO
 2 BALANCE
 3 COUNT              [ COUNT IS CONSTANT OF 1
 4                    [ DUMMY TOTALLING FIELD
*HEAD L,M CH1,3,55
DD/MM/YY       BALANCE ANALYSIS FOR AREA 'MM'      PAGE PPPP

    CUSTOMER NUMBER                                  BALANCE
*OUT L 0,1
      AAAA                                        £222222.22
*OUT M 1,2
 'TOTAL BALANCE FOR AREA' MM                     £2222222.22

 'AVERAGE BALANCE'                               £4444444.44
*DETAB RECORD
C AREA =  '?'   21   37          [ SELECT ON AREA
*
*DETAB BREAK M
A *4   MV *2                     [ SET WORK = TOTAL BALANCE
  *4   /  *3                     [ DIVIDE BY NO. OF RECORDS
```

```
*
*SORT  *M,*A
*STOP
```

As all fields to be printed must be declared in the *INLIST, provision must be made for the average balance, which does not appear on the input file. In fact, this field does not exist until a control break occurs on 'AREA'. For this reason, a dummy totalling field is declared in the *INLIST.

At the RECORD Decision Point, records for areas 21 and 37 are selected.

The field specifying character 3 is of special interest. It has been allocated to the Reserved Word COUNT, which contains a constant value of 1. For each record accepted by *DETAB RECORD, the FSC 3 has the value 1. Because it is a totalling field, *3 consequently contains a count of the number of records processed when the break occurs. This information is, of course, required when the average is calculated.

At the BREAK M Decision Point (that is, change of area), the accumulated total balance (*2) is moved into the dummy totalling field (*4), and then this field is divided by the number of records in the area (*3). Following this processing, FILETAB prints the *OUT M details, using the contents of the fields *M (area), *2 (total balance) and *4 (average balance).

Sorting takes place after record selection, but before BREAK processing.

A sample of the output from this program appears below:

```
03/08/00      BALANCE ANALYSIS FOR AREA  21        PAGE    1

     CUSTOMER NUMBER                                 BALANCE

       1144                                         £1511.20
       1313                                          £151.50
       8643                                          £109.22
       9187                                           £12.50
       9768                                          £200.30
       9827                                          £210.80

  TOTAL BALANCE FOR AREA  21                        £2195.50
  AVERAGE BALANCE                                    £365.92


03/08/00      BALANCE ANALYSIS FOR AREA  37        PAGE    2

     CUSTOMER NUMBER                                 BALANCE

       2100                                           £30.00
       2238                                          £980.22
       4243                                          £591.50
       7173                                          £492.56
       8431                                            £7.51
       9213                                         £1211.50
       9221                                           £90.50
       9298                                            £8.00

  TOTAL BALANCE FOR AREA  37                        £3411.79
  AVERAGE BALANCE                                    £426.47
```

11.7 File Amendment

In this example, the customer master file is updated. Credit codes of `00` and ' ' are to be replaced by a value of `10`. The account numbers from amended records are to be printed. A summary is also output.

```
*PROGRAM TEST7
*OPTION PE,ZS
*FILE   CUSTFILE RL=332  NAME=custfile.txt
*OFILE  CUSTFL2  RL=332  NAME=custfil2.txt
*DICTIONARY
 ACNO     = 4/4       [ ACCOUNT NUMBER
 CREDCODE = 328/2     [ CREDIT CODE
*
*INLIST
 A ACNO
 1
 2
 3 COUNT
*HEAD L CH1,2,55
              ACCOUNTS WITH CORRECTED CREDIT CODES - DD/MM/YY
*OUT L 1,0
       AAAA
*OUT F 3,0
              111111    'RECORDS PROCESSED'

              222222    'RECORDS UNCHANGED'

              333333     'RECORDS UPDATED'
*
*DETAB RECORD
C CREDCODE = '00'      Y   .   ELSE  [ ZERO CREDIT CODE?
  CREDCODE = '  '      .   Y   .     [ BLANK CREDIT CODE?
A CREDCODE MV '10'     X   X   .     [ SET CREDIT CODE TO 10
  WRITE  CUSTFL2 0/0   X   X   X     [ WRITE RECORD AWAY
  IGNORE               .   .   X     [ PASS FORWARD UPD RECS
*
*DETAB BREAK F
I *1 MV RECOUNT
  *2 MV *1
  *2 -  *3
*
*STOP
```

Note: The use of `COUNT` in the `*INLIST` to accumulate the number of records passed through to the `*INLIST`, that is, those not `IGNORE`d at `*DETAB RECORD`. Automatic totalling cannot be used, because actions on records that obey the `ELSE` rule (and are thus `IGNORE`d) would not automatically increment the relevant `*INLIST` field.

The Reserved Word `RECOUNT`, which contains the number of records read from the main file, is not defined in the `*INLIST`. Its value is moved to a dummy totalling field for printing out as the total number of records processed. Sample output is shown below:

```
             ACCOUNTS WITH CORRECTED CREDIT CODES - 20/08/00


     2100
```

```
          9832
                      21     RECORDS PROCESSED
                      19     RECORDS UNCHANGED
                       2      RECORDS UPDATED
```

11.8 File Matching

In this example, transactions on `STRAN` (the sorted `TRANSDY` file) are matched with the customer master file (`CUSTFILE`), so that customer names may be retrieved. In `*DETAB RECORD`, if the first record presented from `CUSTFILE` has a key less than that of the field on `STRAN` with which it is being compared, the `REPEAT` action takes effect; this accesses the next record on `CUSTFILE` and compares it with the relevant field on `STRAN`.

```
*PROGRAM TEST8
*OPTION PE
*FILE STRAN RL=64  NAME=transdy.txt
                    [ TRANSACTION FILE SORTED BY ACCOUNT NO.
*IFILE CUSTFILE BC=YES,RL=332 NAME=custfile.txt
*DICTIONARY
 TACNO     = 4/4        [ ACCOUNT NO. ON STRAN
*
 CUSTIO    = -336:4     [ CUSTFILE DATA WILL BE AT -332/332
 CACNO     = -328/4     [ ACCOUNT NO. ON CUSTFILE
 NAME      = -324/36    [ CUSTOMER NAME ON CUSTFILE
*
 MESG      = -357/21    [ WILL HOLD ERROR MESSAGE
*
*INLIST
 M TACNO
 A NAME
*HEAD L CH1,2,55
DD/MM/YY       TRANSACTION CROSS-REFERENCE         PAGE PPPP



                  ACCOUNT               CUSTOMER

                  NUMBER                NAME
*OUT M 1,0
                  MMMM
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*DETAB START
A READ   CUSTFILE CUSTIO/336          [ GET FIRST RECORD
  MESG   MV '*** NOT ON FILE ***'     [ BUILD MESSAGE
*
*DETAB RECORD
C CUSTIO   = -1                Y  N  N  N   [ EOF ON CUSTFL
  TACNO   ?  CACNO             .  EQ LT GT  [ COMPARE KEYS
A READ  CUSTFILE CUSTIO/336    .  .  .  X   [ GET NEXT RECD
  NAME    MV MESG              X  .  X  .   [ NOT ON FILE
  REPEAT                       .  .  .  X
*
*STOP
```

Note: As '`BC=YES`' appears on the `*IFILE` directive, the record is read into -336/336; the first four bytes (`CUSTIO`) holds the byte count, and the actual data starts at `-332`.

Output from this program appears below.

```
25/08/00         TRANSACTION CROSS-REFERENCE       PAGE    1

                  ACCOUNT         CUSTOMER
                  NUMBER          NAME

                  1144            SMITHS TIMBER LTD
                  1313            R.S. WILDE JOINERY
                  2100            JOHNSONS (DEMOLITION) LTD.
                  2441            MERCANTILE CARRIERS LTD.
                  9221            RICHARDSON & GREEN
                  9298            WATERHOUSE (UK) LTD.
                  9768            MATTHEW AND SON
```

11.9 Updating an Indexed-sequential File

This program shows a simple application in which the Indexed-sequential file `STKREF` is updated with data held on `STKUPD`. `STKREF` contains details of the location of items in a warehouse. The file is indexed on item number, and contains one record for each item stocked.

Transactions on `STKUPD` may be insertions, replacements or deletions. A report is produced showing the effect of each such transaction.

```
*PROGRAM TEST9
*OPTION PE
*FILE   STKUPD RL=21       NAME=stkupd.txt
*IOFILE STKREF RL=21,OR=I  NAME=stkref.idx
*DICTIONARY
 UPDIO    = 0/21       [ I/O AREA FOR STKUPD
 UPDITEM  = 0:4        [ ITEM NUMBER
 UPDCODE  = 4/1        [ INSERT/REPLACE/DELETE
 UPDDESC  = 5/10       [ DESCRIPTION
 UPDBIN   = 15/6       [ BIN LOCATION
 DELFLAG  = -17/1      [ DELETED RECORD IF *
 REFIO    = -21/21     [ I/O AREA FOR STKREF
 MESG     = -30/9      [ ACTION MESSAGE
 MESG2    = -39/9      [ DIAGNOSTIC
 INSCNT   = -44:4      [ COUNT OF INSERTS
 REPCNT   = -48:4      [ COUNT OF REPLACEMENTS
 DELCNT   = -52:4      [ COUNT OF DELETIONS
 REJCNT   = -56:4      [ COUNT OF REJECTIONS
 IRDCODE  = -57/1      [ ACTION CODE
*INLIST
 A IRDCODE
 B UPDITEM
 C UPDDESC
 D UPDBIN
 E MESG
 F MESG2
 1                     [ DUMMY TOTAL - WILL HOLD INSCNT
 2                     [ DUMMY TOTAL - WILL HOLD REPCNT
 3                     [ DUMMY TOTAL - WILL HOLD DELCNT
 4                     [ DUMMY TOTAL - WILL HOLD REJCNT
```

```
*HEAD L CH1,2,55
DD/MM/YY           STOCK LOCATION UPDATE           PAGE PPPP

I/R/D    ITEM    DESCRIPTION  BIN     ACTION       COMMENTS
*OUT L 1,0
A        BBBBBBB CCCCCCCCCC   DDDDDD  EEEEEEEEEE   FFFFFFFFF
*OUT F 3,0
            111  'ITEMS INSERTED'

            222  'ITEMS REPLACED'

            333  'ITEMS DELETED'

            444  'TRANSACTIONS REJECTED
*DETAB START
A ?CNT  Z          INS  REP  DEL   REJ   [ CLEAR COUNTERS
*DETAB RECORD
C UPDCODE  =  '?'         I  R  D        [ INS/REP/DEL
A MESG2    S              X  X  X        [ CLEAR COMMENT
  IRDCODE  MV UPDCODE     X  X  X        [ CODE IN WORK AREA
  UPDCODE  S              X  X  X        [ CLEAR CODE
  CALL     INSERT         X  .  .        [ DO INSERT
  CALL     REPLACE        .  X  .        [ DO REPLACE
  CALL     DELETE         .  .  X        [ DO DELETE
*DETAB INSERT
C LOOKUP   STKREF UPDITEM Y  ELSE        [ ON FILE...
  IF       DELETED        N  .           [ DELETED...
A WRITE    STKREF UPDIO   .  X           [ INSERT
  MESG     MV 'INSERT'    .  X           [ ACTION TAKEN
  INSCNT   +  1           .  X           [ UPDATE COUNTER
  MESG     MV 'NONE'      X  .           [ ERROR
  MESG2    MV 'DUPLICATE' X  .           [  MESSAGES
  REJCNT   +  1           X  .           [ UPDATE COUNTER
*DETAB REPLACE
C LOOKUP   STKREF UPDITEM Y  ELSE        [ ON FILE...
  IF       DELETED        N  .           [ DELETED...
A WRITE    STKREF UPDIO   X  .           [ REPLACE
  MESG     MV 'REPLACE'   X  .           [ ACTION TAKEN
  REPCNT   +  1           X  .           [ UPDATE COUNTER
  MESG     MV 'NONE'      .  X           [ ERROR
  MESG2    MV 'UNKNOWN'   .  X           [  MESSAGES
  REJCNT   +  1           .  X           [ UPDATE COUNTER
*DETAB DELETE
C LOOKUP   STKREF UPDITEM Y  ELSE        [ ON FILE...
  IF       DELETED        N  .           [ DELETED...
A UPDCODE  MV '*'         X  .           [ SET DEL FLAG
  WRITE    STKREF UPDIO   X  .           [ FLAG AS DEAD
  MESG     MV 'DELETE'    X  .           [ ACTION TAKEN
  DELCNT   +  1           X  .           [ UPDATE COUNTER
  MESG     MV 'NONE'      .  X           [ ERROR
  MESG2    MV 'UNKNOWN'   .  X           [ MESSAGES
  REJCNT   +  1           .  X           [ UPDATE COUNTER
*DETAB DELETED
I READ     STKREF REFIO                  [ GET RECORD
C DELFLG   EQ '*'         Y  N           [ DEAD...
A EXIT     ?              T  F           [ GO BACK
*DETAB BREAK F
A *1       MV INSCNT                     [ INSERT
  *2       MV REPCNT                     [ REPLACE
  *3       MV DELCNT                     [ DELETE
  *4       MV REJCNT                     [ REJECT
*STOP
```

The key on STKREF is four bytes long, starting at position one of each record. The fifth character of records on STKREF is reserved as a deletion marker. An * signifies that the record has been deleted, but not yet removed from the file.

Job statistics are accumulated without the use of the tabulator, so dummy totalling fields have been allocated in the *INLIST to facilitate their printing at the end of the run.

*Note: The use of extended entry format in the action-only *DETAB START.*

All processing is controlled by *DETAB RECORD, which CALLs routines to perform the three basic update functions.

The dummy totalling fields are initialised at the BREAK F Decision Point. Sample output appears below.

```
20/12/00           STOCK LOCATION UPDATE           PAGE    1
I/R/D   ITEM      DESCRIPTION    BIN     ACTION    COMMENTS

I       6435530   3 AMP FUSE     AQ702A  INSERT
I       7612068   60W BULB       RB202A  INSERT
R       6530688   75W BULB       DD412B  REPLACE
I       8721221   150W BULB      AR004A  NONE      DUPLICATE
D       6623230                          DELETE
R       6434792   3-WAY JCT      CD141B  NONE      UNKNOWN
I       6384120   2 PIN SCT      ED226A  INSERT

            3      ITEMS INSERTED
            1      ITEMS REPLACED
            1      ITEMS DELETED
            2      TRANSACTIONS REJECTED
```

11.10 File Merging

This program merges the files STRAN1 and STRAN2 to give STRAN, whose format is the same as TRANSDY. Both input files are in the sequence determined by their account numbers. No report is required.

```
*PROGRAM TEST10
*OPTION P,PE
*IFILE  STRAN1  RL=64  NAME=transdy1.txt
*IFILE  STRAN2  RL=64  NAME=transdy2.txt
*OFILE  STRAN   RL=64  NAME=transdy3.txt
*DICTIONARY
AREA1    = -64/64                     [ INPUT AREA FOR STRAN1
KEY1     = -60/4                      [ KEY
*
AREA2    = -128/64                    [ INPUT AREA FOR STRAN2
KEY2     = -124/4                     [ KEY
*
*DETAB START
A READ   STRAN1 AREA1                 [ READ FIRST RECORD
  READ   STRAN2 AREA2                 [  FROM EACH FILE
  GOTO   MERGE
*
*DETAB MERGE
```

```
C AREA1:4 =  -1         Y  Y  N  N  N  N  [ STRAN1 EOF?
  AREA2:4 =  -1         Y  N  Y  N  N  N  [ STRAN2 EOF?
  KEY1 ? KEY2           .  .  .  EQ LT GT [ COMPARE KEYS
A WRITE  STRAN  AREA1   .  .  X  X  X  .  [ OUTPUT RECORD
  WRITE  STRAN  AREA2   .  X  .  X  .  X  [  WITH LOWER KEY
  READ   STRAN1 AREA1   .  .  X  X  X  .  [ READ NEXT RECORD
  READ   STRAN2 AREA2   .  X  .  X  .  X  [  OFF RIGHT FILE
  GOTO   MERGE          .  X  X  X  X  X  [
  CLOSE  STRAN          X  .  .  .  .  .  [ CLOSE O/P FILE
  IGNORE                X  .  .  .  .  .  [ FORCE EOJ
*
*STOP
```

The Fixed Logic is bypassed, because all processing is performed at the START Decision Point. Processing is terminated by an IGNORE (equivalent to EXIT F) being obeyed.

There is no printed output from this example.

11.11 File Updating

This example shows a simple update, in which the value field from transaction records is used to update the outstanding balance on customer records. Transactions for which no customer record exists are written to the file NOMATCH.

```
*PROGRAM TEST11
*OPTION PE
*IFILE  CUSTFL  RL=332  NAME=custfile.txt
*IFILE  STRAN   RL=66   NAME=transdy.txt
*OFILE  CUSTFL2 RL=332  NAME=custfil2.txt
*OFILE  NOMATCH RL=66   NAME=nomatch.txt
*DICTIONARY
*
* TRANSACTION RECORDS AT -66/66
TREC     = -66/66     [ INPUT RECORD
TACCNO   = -62/4      [ ACCOUNT NUMBER
TVALUE   = -40/4      [ VALUE
*
* CUSTOMER RECORDS AT -398/332
CREC     = -398/332   [ OUTPUT RECORD
CACCNO   = -394/4     [ ACCOUNT NUMBER
CBALANCE = -92:4      [ BALANCE
*
*DETAB START
I READ CUSTFL CREC    [ FIRST CUSTOMER
  READ STRAN  TREC    [ FIRST TRANSACTION
C CACCNO ? TACCNO       LT  LT  EQ  EQ  GT  GT  ELSE
* COMPARE KEY
  CREC:4 EQ -1          N   Y   N   N   N   N     .
*      EOF ON CUSTFL?
  TREC:4 EQ -1          N   N   N   Y   N   Y     .
*       EOF ON STRAN?
A CBALANCE + TVALUE     .   .   X   .   .   .     .
*      UPDATE BALANCE
  WRITE NOMATCH TREC    .   X   .   .   X   .     .
*    NO MATCH
  READ STRAN TREC       .   X   X   .   X   .     .
*    NEXT TRANSACTION
```

```
  WRITE CUSTFL2 CREC    X   .   .   X   .   X     .
* COPY CUSTFL
  READ CUSTFL CREC      X   .   .   X   .   X     .
*       GET NEXT CUST
  REPEAT                X   X   X   X   X   X     .
*  LOOP ROUND
  IGNORE                .   .   .   .   .   .     X
*   FINISH PROCESSING
*
*STOP
```

The Fixed Logic is not used in this example. All processing is performed at the `START` Decision Point, and the job is terminated by an `IGNORE` action.

The `STRAN` file was created by sorting the `TRANSDY` file into account number sequence.

`*DETAB START` is written in such a way that more than one transaction can be handled for a single customer.

Note: The use of `REPEAT` to re-enter the condition/action part of `*DETAB START`. This does not perform the initial action, and is more efficient than a `GOTO`.

11.12 Data Input and Validation

This program reads cards containing details of stock location movements. The valid records are written to the file `STKUPD`. *An update program using this file is shown in Updating an Indexed-sequential File on page 11.*

Records must contain a transaction code of `I`, `R` or `D`. Item number must be numeric. Bin location comprises of two alphabetic characters, three numeric and one more alphabetic.

Invalid records are highlighted on the report, and a summary is produced:

```
*PROGRAM TEST12
*OPTION PE,ZS,CCAD
*FILE   CARDSIN        NAME=test12cr.txt
*OFILE  STKUPD  RL=21  NAME=stkupd.txt
*DICTIONARY
INAREA   = CC1-24   [ INPUT RECORD
INCODE   = CC1      [ TRANSACTION CODE - I/R/D
INITEM   = CC2-8    [ ITEM NUMBER      - NUM.
INDESC   = CC9-18   [ DESCRIPTION
INBIN    = CC19-24  [ BIN LOCATION     - AA999A
*
OUTAREA  = -21/21   [ OUTPUT RECORD
OUTITEM  = -21:4    [ ITEM NUMBER
OUTCODE  = -17/1    [ TRANSACTION CODE
OUTDESC  = -16/10   [ DESCRIPTION
OUTBIN   = -6/6     [ BIN LOCATION
*
GOODRECS = -28:4    [ ACCEPTED RECORDS
BADRECS  = -32:4    [ REJECTED RECORDS
ACTION   = -40/8    [ ACTION TAKEN
*
```

```
ERRFLAG  = -54/14   [ ERROR FLAGS
ERRFLAG1 = -54/1    [ CODE
ERRFLAG2 = -53/7    [ ITEM NUMBER
ERRFLAG3 = -46/6    [ BIN LOCATION
*
P1       = -60:4    [ POINTER
P2       = -64:4    [ POINTER
*
VALMASK  = -8/8     [ USED IN VALIDATION
*
*INLIST
A INAREA
B ERRFLAG
C ACTION
2 GOODRECS
3 BADRECS
*HEAD L CH1,3,58
DD/MM/YY                STOCK LOCATION VALIDATION
PAGE PPPP

I/R/D         ITEM         DESCRIPTION         BIN
ACTION

              NO.                              NO.
TAKEN


*OUT L 1,1
  A           AAAAAAA      AAAAAAAAAA          AAAAAA
CCCCCCCC

  B           BBBBBBB                          BBBBBB
*OUT F 3,0
              22222        'RECORDS ACCEPTED'

              33333        'RECORDS REJECTED'
*DETAB RECORD
I ERRFLAG  S                                  [ CLEAR ERRORS
C IF       CODEOK           Y  ELSE           [ CHECK CODE
  IF       ITEMOK           Y  .              [ CHECK ITEM
  IF       BINOK            Y  .              [ CHECK BIN
A GOODRECS MV ?             1  0              [ UPDATE
  BADRECS  MV ?             0  1              [  COUNTERS
  ACTION   MV 'ACCEPTED'    X  .              [ ACTION
  ACTION   MV 'REJECTED'    .  X              [  TAKEN
  OUTCODE  MV INCODE        X  .              [ BUILD
  OUTITEM  MV INITEM        X  .              [  OUTPUT
  OUTDESC  MV INDESC        X  .              [  RECORD
  OUTBIN   MV INBIN         X  .              [  AND
  WRITE    STKUPD OUTAREA   X  .              [  OUTPUT
*
*DETAB CODEOK
C INCODE   EQ '?'           I  R  D  ELSE     [ CHECK CODE
A ERRFLAG1 MV '*'           .  .  .  X        [ HIGHLIGHT
  EXIT     ?                T  T  T  F        [ RETURN
*
*DETAB ITEMOK
I VALMASK  MV '0000000"'                      [ VALIDATE
  P1       MV A'INITEM                        [  THAT ITEM
C IF       VALID            Y  N              [  IS NUMERIC
A ERRFLAG2 MV '*******'     .  X              [ HIGHLIGHT
```

```
  EXIT     ?                T  F           [ RETURN
*
*DETAB BINOK
I VALMASK  MV 'AA000A"'                    [ VALIDATE
  P1       MV A'INBIN                      [  BIN NUMBER
C INCODE   EQ 'D'          N  N  ELSE      [  IF NOT
  IF       VALID           Y  N  .         [  DELETION
A ERRFLAG3 MV '******'     .  X  .         [ HIGHLIGHT
  EXIT     ?               T  F  T         [ RETURN
*
*     ************************************
*     *** STANDARD VALIDATION ROUTINE *** *
*     ************************************
*
* THIS ROUTINE VALIDATES THE FIELD ADDRESSED BY P1.
* INPUT PARAMETER IS VALMASK TERMINATED BY ".
*     0 - NUMERIC
*     A - ALPHABETIC
*     N - ALPHANUMERIC
* " IN FIELD BEING VALIDATED TERMINATES OPERATION.
*
*DETAB VALID
I P2       MV A'VALMASK
C (P1)/1   EQ '"'     N  N  N  N  N  N  N  N  Y  N  ELSE
  (P2)/1   EQ '"'     N  N  N  N  N  N  N  N  .  Y  .
  (P2)/1   EQ '?'     0  A  A  A  N  N  N  N  .  .  .
  (P1)/1   GE '0'     Y  .  .  .  Y  .  .  .  .  .  .
  (P1)/1   LE '9'     Y  .  .  .  Y  .  .  .  .  .  .
  (P1)/1   GE 'A'     .  Y  .  .  .  Y  .  .  .  .  .
  (P1)/1   LE 'I'     .  Y  .  .  .  Y  .  .  .  .  .
  (P1)/1   GE 'J'     .  .  Y  .  .  .  Y  .  .  .  .
  (P1)/1   LE 'R'     .  .  Y  .  .  .  Y  .  .  .  .
  (P1)/1   GE 'S'     .  .  .  Y  .  .  .  Y  .  .  .
  (P1)/1   LE 'Z'     .  .  .  Y  .  .  .  Y  .  .  .
A P1       +  1       X  X  X  X  X  X  X  X  .  .  .
  P2       +  1       X  X  X  X  X  X  X  X  .  .  .
  REPEAT              X  X  X  X  X  X  X  X  .  .  .
  EXIT     ?          .  .  .  .  .  .  .  .  T  T  F
*
* ****** END OF STANDARD VALIDATION ROUTINE ******
*
*GO
```

The main attraction of this program is the use of a standard data validation routine which could be stored on a copy file. Because alphabetic characters are not consecutive in the collating sequence, validation may be quite complex.

When the transaction code is `D`, bin number is not present.

Note: The use of the `IF` verb to structure the program.

A sample report is shown below:

```
19/12/00          STOCK LOCATION VALIDATION        PAGE    1

I/R/D     ITEM      DESCRIPTION         BIN          ACTION
           NO.                           NO.           TAKEN


  I     6435530      3 AMP FUSE         AQ702A      ACCEPTED
```

```
 I     7612068      60W BULB          RB202A      ACCEPTED

 R     6530688      75W BULB          DD412B      ACCEPTED

 R     9525248      3 CORE CAB        C5674V      REJECTED
                                      ******

 I     8721221      150W BULB         AR004A      ACCEPTED

 D     6623230                                    ACCEPTED

 R     6434792      3-WAY JCT         CD141B      ACCEPTED

 I     6384120      2 PIN SCT         ED226A      ACCEPTED


              7   RECORDS ACCEPTED
              1   RECORDS REJECTED
```

11.13 Totalling Using *TABLE

This program reads the TRANSDY file and produces a summary file (SUMMARY). The summary file contains one record for each month, and holds totals of value, VAT and discount.

```
*PROGRAM TEST13
*OPTION PE,ZS,K
*FILE TRANSDY RL=64   NAME=transdy.txt
*OFILE SUMMARY RL=14  NAME=summary.txt
*DICTIONARY
 MONTH    = 22/2     [ MONTH
 VALUE    = 28:4     [ VALUE
 VAT      = 32:4     [ VAT
 DISC     = 36:4     [ DISCOUNT
*
 IOAREA   = -14/14   [ USED BY MONTHS AND SUMMARY
 IOMONTH  = -14/2    [ MONTH
IOVALUE   = -12:4    [ TOTAL VALUE
 IOVAT    = -8:4     [ TOTAL VAT
 IODISC   = -4:4     [ TOTAL DISCOUNT
*
*INLIST
 1 COUNT
*OUT F CH1,0
     11111  'RECORDS PROCESSED ON TRANSDY'
*TABLE MONTHS
KK DDDDDDDDDDDD
01
02
03
04
05
06
07
08
09
10
11
```

```
12
*DETAB RECORD
C LOOKUP   MONTHS MONTH    Y       [ LOCATE ENTRY
A READ     MONTHS IOAREA   X       [ RETRIEVE ENTRY
  GOTO     SUM             X       [ UPDATE
*DETAB SUM
C IOVALUE/4 =   ' '        Y  N    [ UNUSED
A IOVALUE   Z              X  .    [ INITIALISE
  IOVAT     Z              X  .    [  DATA
  IODISC    Z              X  .    [  PORTION
  IOVALUE   +  VALUE       X  X    [ TOTAL VALUE
  IOVAT     +  VAT         X  X    [ TOTAL VAT
  IODISC    +  DISC        X  X    [ TOTAL DISCOUNT
  WRITE MONTHS IOAREA      X  X    [ REWRITE ENTRY

*DETAB BREAK F
I IOMONTH  MV '01'                 [ FIRST KEY
C LOOKUP   MONTHS  IOMONTH Y       [ POSITION AT START
A READ     MONTHS  IOAREA  X       [ GET ENTRY
  CALL     ZEROISE         X       [ REPLACE BLANKS
  WRITE    SUMMARY IOAREA  X       [ WRITE AWAY SUMMARY
  GOTO     BREAKF2         X       [ GET NEXT ENTRY
*
*DETAB BREAKF2
I READ      MONTHS  IOAREA         [ GET ENTRY
C IOMONTH   = '12'         Y  N    [ SEE IF LAST MONTH
A CALL      ZEROISE        X  X    [ REPLACE BLANKS
  WRITE    SUMMARY IOAREA  X  X    [ WRITE AWAY SUMMARY
  READ MONTHS IOAREA       X  X    [ GET NEXT ENTRY
  REPEAT                   .  X    [ LOOP ON ROUND
*
*DETAB ZEROISE
C IOVALUE/8 = ' '          Y  N    [ UNUSED ENTRY
A IOVALUE Z                X  .    [ REPLACE
  IOVAT  Z                 X  .    [  BLANKS
  IODISC Z                 X  .    [  BY ZEROS
  EXIT    T                X  X    [ RETURN
*
*GO
```

The lookup table `MONTHS` has one entry for each month of the year. The month number forms the key, and each descriptor comprises three four byte (binary) totals: value, VAT, and discount, respectively. These descriptors are originally blanks, and therefore require initialisation.

Note: The use of `*OPTION K` to return the key to `IOAREA` at `*DETAB RECORD`.

At the end of run, at the `BREAK F` Decision Point, the contents of this table are output to the summary file.

The only output is a count of the records processed.

11.14 Totalling using Arrays and Pointers

The purpose of this program is identical to that of the example in *Totalling Using *TABLE on page 18*. It does, however, tackle the problem in a different way, by using an array of totals that are addressed by Pointers.

```
*PROGRAM TEST14
*OPTION PE,ZS
*FILE  TRANSDY  RL=64  NAME=transdy.txt
*OFILE SUMMARY  RL=14  NAME=summary.txt
*
*DICTIONARY
TRANS-REC   = 0/64
MONTH       = TRANS-REC+22/2
VALUE       = TRANS-REC+28:4
VAT         = TRANS-REC+32:4
DISC        = TRANS-REC+36:4
SUMY-REC    = -14/14
IOAREA      = SUMY-REC+0/14
IOMONTH     = SUMY-REC+0/2
IOTOTALS    = SUMY-REC+2/12
*DICTIONARY
COUNTER     = -20:4                 [ WORK COUNTER
P1 = -24:4                          [ POINTER
ARRAYEL = -168:4                    [ FIRST ELEMENT OF ARRAY
*INLIST
1 COUNT
*OUT F
     11111            'RECORDS PROCESSED ON TRANSDY'
*
*DETAB START
I P1        MV A'ARRAYEL             [ ADDRESS OF ARRAY
  COUNTER   Z
C COUNTER   =  12    N  Y
A (P1):8+   Z        X  .
  COUNTER   +  1     X  .
  REPEAT             X  .
*
*DETAB RECORD
A P1       MV MONTH                  [ USE MONTH NUMBER
  P1        *  12                    [    TO STEP TO
  P1        -  12                    [    APPROPRIATE PART
  P1        +  A'ARRAYEL             [    OF THE ARRAY
  (P1):4+   +  VALUE                 [ VALUE FOR THE MONTH
  (P1):4+   +  VAT                   [ VAT FOR THE MONTH
  (P1):4    +  DISC                  [ DISCOUNT FOR THE MTH
*DETAB BREAK F
I P1        MV A'ARRAYEL             [ ADDRESS OF ARRAY
  COUNTER   MV 1                     [ MONTH COUNTER
C COUNTER   GT  12          Y  N     [ FINISHED...
A IOMONTH   MV  COUNTER     .  X     [ GET MONTH NUMBER
  COUNTER   +   1           .  X     [ UPDATE MONTH
  IOTOTALS  MV  (P1)/12+    .  X     [ EXTRACT TOTALS
  WRITE     SUMMARY IOAREA  .  X     [ WRITE SUMMARY RECORD
  REPEAT                    .  X     [ LOOP ON ROUND
*
*GO
```

Note: The programmer has explicitly specified fields by means of the `*DICTIONARY` directive.

An array is used to hold the accumulated totals of value, VAT and discount for each month. The totals appear in this sequence for each month, so that the three totals for month `01` are followed by month `02` totals and so on.

Note: No operation can be performed on a binary field longer than eight bytes. To initialise the complete `ARRAYEL` with binary zeros, a

repetitive operation is performed at `*DETAB START`, with a condition to end repetition at the appropriate point.

The only output is a count of the records processed.

11.15 N-up Printing - Different Items

This example forms a basis for 'N-up' label print programs. In this program, labels are produced 3-up. The information is taken from the name and address fields on customer records.

```
*PROGRAM TEST15
*OPTION PE,ZS
*FILE   CUSTFL RL=332  NAME=custfile.txt
*DICTIONARY
NAMADD  = 8/140            [ NAME/ADDRESS ON RECORD
*
STORE1   = -140/140        [ STORE FOR FIRST LABEL
STORE2   = -280/140        [ STORE FOR SECOND LABEL
FLAG     = -281/1          [ 3-UP INDICATOR
*INLIST
A STORE1
B STORE2
C NAMADD
*OUT L 4,0
AAA----------AAA      BBB----------BBB      CCC----------CCC

AAA-----AAA           BBB-----BBB           CCC-----CCC

AAA-----AAA           BBB-----BBB           CCC-----CCC

AAA-----AAA           BBB-----BBB           CCC-----CCC

AAAAAAAA              BBBBBBBB              CCCCCCCC
*DETAB START
A FLAG     MV '1'           [ INITIALISE 3-UP INDICATOR
*
*DETAB RECORD
C FLAG     =  '?'        1  2  3    [ FIRST/SECOND/THIRD
  STORE?   MV NAMADD     1  2  .    [ STORE N/A
A FLAG     MV '?'        2  3  1    [ UPDATE FLAG
  IGNORE                 X  X  .    [ ONLY PRINT ON THIRD
*
*DETAB ENDFILE
C FLAG     =  '?'        1  2  3    [ CHECK FOR RESIDUE
A FLAG     MV '3'        .  X  .    [ FORCE OUTPUT
  STORE2   S             .  X  .    [ BLANK OUT
  NAMADD   S             .  X  X    [  IRRELEVANT LABELS
*
*GO
```

The main purpose of the processing at the `RECORD` Decision Point is to ensure that printing takes place only when three labels are available. For the first two records of the group, the data is stored away and printing suppressed by the `IGNORE` action. Printing takes place for every third record.

When end-of-file is reached, there may be some labels which have yet to be printed. This situation occurs when the number of records on file is not

a multiple of 3. Processing at the ENDFILE Decision Point checks how many labels are still to be printed. Store areas are blanked out to prevent duplication of the previous batch of labels. Control passes to the RECORD Decision Point only if there is still printing to be performed.

Note: No example output is given. This is due to the limited width available.

11.16 Enclosing Negative Values in Brackets

This example allows all negative totalling fields to be enclosed in brackets:

```
*PRO TEST16
*FILE STARFILE  NAME=test16cr.txt
*DICT
R-R = 0/1 , Q-Q = 1/1 , P-P = 2/1 , O-O = 3/1 , N-N = 4/1
M-M = 5/1 , VAL1 = 80:4 , VAL2 = 84:4 , VAL3 = 88:4
OBRAC = 100/4 , CBRAC = 120/4
*INL
R R-R , Q Q-Q , P P-P , O O-O , N N-N , M M-M , ( OBRAC , )
CBRAC
1 VAL1 , 2 VAL2 , 3 VAL3 , 4
*DETAB RECORD
I VAL1 Z
  VAL2 Z
  VAL3 Z
C  6/1 = '-'   Y  N  .  .  .  . ELSE
  12/1 = '-'   .  .  Y  N  .  .  .
  18/1 = '-'   .  .  .  .  Y  N  .
A VAL1 -  7/5  X  .  .  .  .  .  .
  VAL1 +  7/5  .  X  .  .  .  .  .
  VAL2 - 13/5  .  .  X  .  .  .  .
  VAL2 + 13/5  .  .  .  X  .  .  .
  VAL3 - 19/5  .  .  .  .  X  .  .
  VAL3 + 19/5  .  .  .  .  .  X  .
*HEAD L CH1,2
HEADING              DD/MM/YY                      PAGE PPPP
*OUT L 1,0
 R Q P O N M (111111) (222222) (333333) (444444)
*OUT M 1,0
           M (111111) (222222) (333333) (444444)
*OUT N 1,0
         N   (111111) (222222) (333333) (444444)
*OUT O 1,0
       O     (111111) (222222) (333333) (444444)
*OUT P 1,0
     P       (111111) (222222) (333333) (444444)
*OUT Q 1,0
   Q         (111111) (222222) (333333) (444444)
```

```
*OUT R 1,0
 R           (111111) (222222) (333333) (444444)
*OUT F 1,0
'FINAL'      (111111) (222222) (333333) (444444)
*DETAB BREAK L,M,N,O,P,Q,R,F
I *4 MV *1
  *4 +  *2
  *4 +  *3
  OBRAC S
  CBRAC S
C *? LT 0             1  2  3  4  ELSE
A OBRAC+?/1 MV '('    0  1  2  3  .
  CBRAC+?/1 MV ')'    0  1  2  3  .
  *( MV OBRAC         X  X  X  X  X
  *) MV CBRAC         X  X  X  X  X
*GO
```

Sample output from this example is as follows:

```
HEADING              17/07/00                     PAGE    1


 0 0 0 0 0 0 ( 49088)   32639  (   3599) ( 20048)
           0 ( 49088)   32639  (   3599) ( 20048)
 0 0 0 0 0 1 ( 32897)   31611    55255    53969
           1 ( 32897)   31611    55255    53969
         0   ( 81985)   64250    51656    33921
 0 0 0 0 1 0   31611  ( 42149)   55512    44974
           0   31611  ( 42149)   55512    44974
         1     31611  ( 42149)   55512    44974
       0     ( 50374)   22101   107168    78895
 0 0 0 1 0 0 ( 42149)   27756  (   2314) ( 16707)
           0 ( 42149)   27756  (   2314) ( 16707)
         0   ( 42149)   27756  (   2314) ( 16707)
       1     ( 42149)   27756  (   2314) ( 16707)
     0       ( 92523)   49857   104854    62188
 0 0 1 0 0 0   27756  ( 44976) (   2057) ( 19277)
           0   27756  ( 44976) (   2057) ( 19277)
         0     27756  ( 44976) (   2057) ( 19277)
       0       27756  ( 44976) (   2057) ( 19277)
     1         27756  ( 44976) (   2057) ( 19277)
   0         ( 64767)    4881   102797    42911
 0 1 0 0 0 0   20560    32125  (   1800)   50885
           0   20560    32125  (   1800)   50885
         0     20560    32125  (   1800)   50885
```

```
        0       20560    32125  (  1800)   50885
      0         20560    32125  (  1800)   50885
    1           20560    32125  (  1800)   50885
  0           ( 44207)   37006   100997    93796
  1 0 0 0 0 0 ( 33411) ( 45747) (  1543) ( 80701)
            0 ( 33411) ( 45747) (  1543) ( 80701)
          0   ( 33411) ( 45747) (  1543) ( 80701)
        0     ( 33411) ( 45747) (  1543) ( 80701)
      0       ( 33411) ( 45747) (  1543) ( 80701)
    0         ( 33411) ( 45747) (  1543) ( 80701)
  1           ( 33411) ( 45747) (  1543) ( 80701)
  FINAL       ( 77618) (  8741)   99454    13095
```

11.17 Reading an I-S File using an Alternate Index

The Indexed-sequential file in this example has a main key at byte zero for sixteen bytes. It has three alternate keys at byte 73 for four bytes, byte 74 for three bytes and byte 40 for fifteen bytes.

```
*PRO TEST17
*IFILE IFILE OR=I NAME=altindex.idx PK=0/16 AK1=73/4
AK2=74/3 AK3=40/15 AD3=Y      [ ACCESS FILE BY ALTERNATE KEY
*DICT
IKEY = 0/16 , IREC = 0/80 , TORF = 80/6
X-T  =  /6  , X-F     =  /6  , X-D  =   /6
*DETAB START
I IREC MV 'OPTPSTBY'
  X-T MV 'TRUE  '
  X-F MV 'FALSE '
  X-D MV 'DUPL  '
C LOOKUP  IFILE IKEY      Y  N  [ LOOKUP FILE WITH MAIN KEY
A READ    IFILE IREC      X  .  [ READ RECORD IF LOOKUP TRUE
  TORF MV X-?             T  F  [ MOVE TRUE/FALSE
  DISPLAY IREC            X  X  [ DISPLAY RECORD
  DISPLAY TORF            X  X  [ DISPLAY TRUE/FALSE
  CLOSE IFILE             X  X  [ CLOSE THE FILE
  GOTO START0             X  X
*DETAB START0                    [ MAIN KEY - BYTE 0/16
I IREC MV 'OPTPSTBY'             [ USING MAIN KEY I.E.
  INDEX   IFILE 0                [ (ALTERNATE) KEY
C LOOKUP  IFILE IKEY      Y  N  [ NUMBER ZERO
A READ    IFILE IREC      X  .  [ READ RECORD IF LOOKUP TRUE
  TORF MV X-?             T  F  [ MOVE TRUE/FALSE
  DISPLAY IREC            X  X  [ DISPLAY RECORD
  DISPLAY TORF            X  X  [ DISPLAY TRUE/FALSE
  GOTO START1             X  X
*DETAB START1                   [ ALTERNATE KEY 1-BYTE 73/4
I IREC S                        [ SET UP ANOTHER
  IREC+73/4 MV ' 170'           [ ALTERNATE KEY
  INDEX   IFILE 1               [ & SWITCH TO IT
C LOOKUP IFILE IREC+73/4  Y  N  [ LOOKUP WITH ALTERNATE KEY
A READ    IFILE IREC      X  .  [ READ IF LOOKUP TRUE
  TORF MV X-?             T  F  [ MOVE TRUE/FALSE
  DISPLAY IREC            X  X  [ DISPLAY RECORD
  DISPLAY TORF            X  X  [ DISPLAY TRUE/FALSE
```

```
  GOTO START2             X  X
*DETAB START2                  [ ALTERNATE KEY - BYTE 74/3
I IREC S                       [ SET UP ANOTHER
  IREC+74/3 MV '655'           [ ALTERNATE KEY
  INDEX   IFILE 2              [ & SWITCH TO IT
C LOOKUP  IFILE IREC+74/3 Y  N  [ LOOKUP WITH ALTERNATE KEY
A READ    IFILE IREC      X  .  [ READ IF LOOKUP TRUE
  TORF MV X-?             T  F  [ MOVE TRUE/FALSE
  DISPLAY IREC            X  X  [ DISPLAY RECORD
  DISPLAY TORF            X  X  [ DISPLAY TRUE/FALSE
  GOTO START3             X  X
*DETAB START3                  [ ALTERNATE KEY - BYTE 40/15
I IREC S                       [ SET UP ANOTHER
  IREC+40/15 MV '(:8)'         [ ALTERNATE KEY & SWITCH TO
  INDEX   IFILE 3              [ IT. THIS
  ZREPLY    MV 1               [ ONE HAS DUPLICATES

C ZREPLY     = ?           1  1  0 -101
                                [ ZREPLY = 1 LOOKUP
                                [ AND READ
  LOOKUP IFILE IREC+41/15   Y  N  .  .
                                    [ = 0 NO MORE TO READ
A READ   IFILE IREC         X  .  .  X
                                    [ = -101 READ DUPLICATE
      TORF MV X-?           T  F  .  D
                                    [ MOVE TRUE/FALSE/DUPL
      DISPLAY IREC          X  X  .  X
                                    [ DISPLAY RECORD
      DISPLAY TORF          X  X  .  X
                                  [ DISPLAY TRUE/FALSE/DUPL
      REPEAT                X  .  .  X
                                    [ CHECK FOR DUPLICATE
      CLOSE   IFILE         .  .  X  .
                                    [ ZREPLY = 0 IF NO MORE
      IGNORE                .  .  X  .
                                    [ DUPLICATES
*GO
```

The fixed logic is not used in this example. All processing is performed at the `START` Decision Point and the job is terminated by an `IGNORE` action.

In `*DETAB START` the file is accessed in the normal way using the main key.

In `*DETAB START0` the file is accessed using an alternate key which happens to be the main key.

In `*DETAB START1` and `*DETAB START2` the file is accessed using alternate keys which do not have duplicates.

In `*DETAB START3` the file is accessed using an alternate key which allows duplicates. In this case there is an `INDEX` and a `LOOKUP` followed by one or many `READ`s.

11.18 Updating an I-S File using an Alternate Index

The Indexed-sequential file used in this example is the same one used in the example in *Reading an I-S File using an Alternate Index on page 24*.

```
*PRO TEST18
*IOFILE IFILE OR=I NAME=altindex.idx PK=0/16 AK1=73/4
AK2=74/3 AK3=40/15 AD3=Y       [ ACCESS FILE BY ALTERNATE KEY
*DICT
IKEY     =   0/16 , IREC    =   0/80
IREC-D   =  56/8  , IREC-T  =  64/8   , TORF = 80/6
SVSCLRES = 100:4  , MOD1    = 104:4   , X-W  =   /6
X-T      =    /6  , X-F     =    /6   , X-D  =   /6
*
*DETAB START
I X-T  MV 'TRUE  '
  X-F  MV 'FALSE '
  X-D  MV 'DUPL R'
  X-W  MV 'DUPL W'
  MOD1 MV 1
  IREC MV 'OPTPSTBY'                   [ MAIN KEY IS
C MOD1 = ?             1   1   2   2  [ DEFAULT KEY
  SVSCLRES = -100      .   .   Y   N  [ -100 DUP ALT KEY (W)
  LOOKUP IFILE IREC/16  Y   N   .   .
                                    [ LOOKUP BY ALTERNATE KEY
A READ  IFILE  IREC     X   .   .   .  [ READ IF ALOOKUP TRUE
  DELETE  IFILE         X   .   .   . [ DELETE IF LOOKUP TRUE
  TORF    MV X-?        T   F   W   T
                                     [ MOVE TRUE/FALSE/DUPL W
  DISPLAY IREC          X   X   X   X  [ DISPLAY RECORD
  DISPLAY TORF          X   X   X   X  [ DISPLAY TRUE/FALSE/
                                       [ DUPL W
  IREC    MV 'OPTPSTBY' .   X   .   .  [ GENERATE NEW RECORD
  IREC-D  MV DATE       .   X   .   .  [ SET UP DATE
  IREC-T  MV TIME       .   X   .   .  [ SET UP TIME
  IREC+40/4 MV '(:8)'   .   X   .   .  [ SET UP ALT KEY1
  IREC+73/4 MV '0085'   .   X   .   .  [ SET UP ALT KEY2
  WRITE  IFILE IREC     .   X   .   .  [ WRITE NEW RECORD
  SVSCLRES MV ZREPLY    .   X   .   .  [ SAVE ZREPLY
  MOD1 + 1              .   X   .   .  [ INCREMENT MOD1
  CLOSE IFILE           .   .   X   X  [ CLOSE FILE
  REPEAT                X   X   .   .  [ LOOP BACK
  GOTO START1           .   .   X   X
*DETAB START1                    [ ALTERNATE KEY-BYTE 40/15
I IREC S                         [ SET UP ALTERNATE
  IREC+40/15 MV '(:8)'           [ KEY - DUPLICATES PRESENT
  INDEX   IFILE 3                [ & SWITCH TO IT
  SVSCLRES    MV 1
C SVSCLRES   = ?       1   1 -101  0  [ -101 DUP ALT KEY (R)
  LOOKUP IFILE IREC+41/15   Y   N   .   .
                                    [ LOOKUP BY ALTERNATE KEY

A READ    IFILE IREC    X   .   X   .  [ READ IF LOOKUP TRUE
  SVSCLRES MV ZREPLY    X   .   X   .  [ SAVE ZREPLY
  TORF     MV X-?       T   F   D   .
                                     [ MOVE TRUE/FALSE/DUPL R
  IREC-D   MV DATE      X   .   X   .  [ SET UP DATE
  IREC-T   MV TIME      X   .   X   .  [ SET UP TIME
  DISPLAY  IREC         X   X   X   .  [ DISPLAY RECORD
  DISPLAY  TORF         X   X   X   .
                                  [ DISPLAY TRUE/FALSE/DUPL R
  WRITE   IFILE IREC    X   .   X   .  [ REWRITE RECORD
  REPEAT                X   .   X   .  [ LOOP BACK
  CLOSE   IFILE         .   X   .   X  [ CLOSE THE FILE
  IGNORE                .   X   .   X  [ END RUN
*GO
```

The fixed logic is not used in this example. All processing is performed at the START Decision Point and the job is terminated by an IGNORE action.

In *DETAB START the file is accessed using an alternate key which happens to be the main key. If the record exists it is read and deleted. If the record does not exist a new record is created and written away. ZREPLY is checked to determine if the record has duplicated another alternate key.

In *DETAB START1 the file is accessed using an alternate key which allows duplicates. In this case there is an INDEX and a LOOKUP followed by one or many READs.

Appendix A
Operations Guide

A.1 Overview

The *Operations Guide* provides instructions for installing the FILETAB system software and making it available to developers. It also describes how to compile and run your programs.

A.1.1 What Your Operations Guide Contains

The *Operations Guide* is divided into the following sections:

A.1 Overview — Describes what is available in this document and how it relates to the FILETAB software

A.2 UNIX and Linux Operations Guide — Looks at a UNIX and Linux FILETAB installation concentrating on:

- **Installation** - explains the installation procedure of FILETAB
- **Directory Structure and Environment Variables** - explains the directory structure of the FILETAB installation
- **Compiling and Running Programs** - describes the methods for the compiling and running of programs
- **Result Codes** - lists the result codes used in FILETAB

A.3 Windows NT Operations Guide — Looks at a Windows NT installation concentrating on the same areas as above

A.2 UNIX and Linux Operations Guide

A.2.1 Installation

The steps required to install FILETAB on Linux or UNIX depend on the delivery mechanism and format. These are:

Delivery mechanism	Format	Steps required
File (Download)	gzipped tar	1 Unzip the tar file, see *Unzip the tar File (release_name.tar.gz) on page 3* 2 Retrieve the installation script, see *Retrieve the Installation Script on page 3* 3 Run the installation script, see *Run the Installation Script on page 3*
File (Download)	rpm	RPM Installation, see *RPM Installation on page 2*
Media (CDROM)	tar	1 Retrieve the installation script, see *Retrieve the Installation Script on page 3* 2 Run the installation script, see *Run the Installation Script on page 3*

To carry out any of these steps you must be logged in as a user with appropriate (write) permissions to the directories being modified; normally `/opt` and optionally `/usr/bin /usr/lib` and `/usr/include`.

A.2.1.1 RPM Installation

You can use Gnome RPM (`GnoRPM`) to select and install from the downloaded `rpm` file. Alternatively you can install using `rpm` with:

```
rpm -ivh release_name.rpm
```

With both of these methods FILETAB will be installed under `/opt/filetab` and will be configured for no backward compatibility (VME only), and using the currently installed (and supported) databases. See *Reconfiguring FILETAB on page 4* if you want to reconfigure FILETAB or *Linking FILETAB into the /usr Hierarchy on page 3* if you want to link into the `/usr` hierarchy.

A.2.1.2 Unzip the tar File (release_name.tar.gz)

gunzip `release_name.tar`

or this can be combined with retrieving the installation script:

gunzip `-c release_name.tar.gz | tar -xvf - ./ftcinstall`

gunzip is available from the Free Software Federation at `http://www.gnu.org`

A.2.1.3 Retrieve the Installation Script

tar `xvf release_name.tar ./ftcinstall`

or for a media delivery:

tar `xvf device_name ./ftcinstall`

A.2.1.4 Run the Installation Script

`./ftcinstall`

The shell script `ftcinstall` is supplied to facilitate the installation of FILETAB. It performs the following tasks (offering defaults where appropriate):

(1) Prompts for the installation directory. You should normally leave this as the default (`/opt/filetab`).

(2) Retrieves the software from the supplied tarfile/media into the installation directory.

(3) Creates sample login script extracts to set the environment variables required by FILETAB. It also modifies the compilation script, `ftc`, to set the default variable values.

To do this **ftcinstall** prompts for:

- Backwards compatibility requirements (if any) so that the compiler can handle supported features from other FILETABs
- Paths to any supported libraries if they cannot be found in their default installation directories; for example, for Informix C-ISAM, if `libisam.a` cannot be found in `/usr/lib`
- Asks if the installation should be linked to the `/usr` hierarchy. Doing this will enable users to run FILETAB without modifying their PATH variable, see *A.2.1.5 Linking FILETAB into the /usr Hierarchy* below

A.2.1.5 Linking FILETAB into the /usr Hierarchy

This will enable users to run FILETAB without modifying their `PATH` or `LD_LIBRARY_PATH` environment variables. This can be done (assuming FILETAB was installed into `/opt/filetab`) as follows:

```
cd /opt/filetab; ./ftcinstall -l
```

A.2.1.6 Reconfiguring FILETAB

FILETAB can be automatically reconfigured for a different backward compatibility or if a new (supported) database is installed by using:

(1) For the default (VME) compatibility:

ftcinstall `-v`

(2) For UNITAB compatibility:

ftcinstall -u

For details on manual configuration see the FILETAB Developers Guide.

A.2.2 Additional Steps Before Using FILETAB

Before using FILETAB you may need to carry out the following steps:

(1) Add the FILETAB binary directory (for example: `/opt/filetab/bin`) into users' `PATH` variable.

(2) Add the FILETAB library directory (for example: `/opt/filetab/lib`) into users' `LD_LIBRARY_PATH` variable, or add it into `/etc/ld.so.conf` and run `ldconfig`.

(3) Also add the directories for data access libraries (such as ODBC) into `LD_LIBRARY_PATH` variable, or `/etc/ld.so.conf` file.

A.2.3 To Remove FILETAB from Your System

If you installed FILETAB using RPM then you should use RPM to uninstall it. Otherwise use:

```
ftcuninstall
```

A.2.4 Directory Structure and Environment Variables

The FILETAB for UNIX software components will be installed into three subdirectories of a FILETAB `root` directory. The name of the root directory is held in the environment variable `FTC_ROOT`. The names of the three subdirectories are held in three environment variables:

`BINDIR`	FILETAB executables	(`%FTCDIR%\BIN`)
`LIBDIR`	FILETAB libraries	(`%FTCDIR%\LIB`)
`INCDIR`	FILETAB include files	(`%FTCDIR%\INCLUDE`)

Additionally `LD_LIBRARY_PATH` should contain the `libraries` directory.

A.2.4.1 Executables

The FILETAB for the executables are **`ftcomp`** and **`ftc`**.

A.2.4.1.1 FILETAB Compiler

The FILETAB compiler (`ftcomp`) compiles FILETAB modules producing intermediate C source code, which can then be compiled and linked to produce executable programs. The **`ftcomp`** command uses the following environment variables:

`INCDIR, LIBDIR, FTC_COB, FTC_CC, FTC_FLAGS, FTC_CFLAGS, FTC_LDFLAGS`

`INCDIR`	Can contain a list of directories, each separated by a colon (:), which will be searched for the initialisation files (see the following list). The current working directory is always searched first
`FTC_CC`	Specifies the C compilation command if it is not `cc`
`FTC_COB`	Specifies the COBOL compilation command if it is not **`cob`**
`FTC_FLAGS`	May be used to specify flags for the **`ftcomp`** command, for example -**`D ORACLE`** for the Oracle interface
`FTC_CFLAGS`	May be used to specify flags that are passed to the C compilation command
`FTC_LDFLAGS`	May be used to specify linked flags
`ftcomp`	takes the following arguments:

`ftcomp` command: **`ftcomp`** `[options] file cerrorsfile`

`file` is the name of the FILETAB source to be compiled, `options` can consist of one or more of the following:

`-i Include Path`	A list of directories, each separated by a colon (:), which will be searched instead of those specified in `INCDIR` as above.
`-r Initial-Files`	A list of initialisation files, each file separated by a colon (:). The first must be the configuration file (see below). Following the configuration file can be a list of FILETAB source files which are evaluated before the specified source is compiled. These can be used, for example, to set ‘default’ options. The default Initial-Files are: `ftcconfg.h:.ftcrc`
`-c Target C File`	The intermediate C source (`.c`) file which is produced by the compiler, default: `ftctargt.c`

`-h Target H File`	The include (`.h`) file which is produced by the compiler, default: `ftctargt.h`
`-D flag`	Defines a conditional compilation flag
`-o object-file`	Specifies the name of the executable file to be produced and also that **`cc`** or **`cob`** are called to produce it The **`cob`** command is used to link FILETAB modules that contain `OR=I` files as they must be linked with the Micro-Focus COBOL file handler and run-time system.
`cerrorsfile`	Any errors detected by `cc` or `cob` are placed in `cerrorsfile`

As well as the two target files mentioned **`ftcomp`** sends source and symbol table listings to `stdout`. Any compilation errors and warnings are sent to `stderr`.

A.2.4.1.2 ftc Shell Script

command: **`ftc`** `[-g] file . . .`

This calls the **`ftcomp`** command to compile the specified FILETAB source into C source code and then call either `cc` or `cob` to compile and link the C to produce executable code.

The defaults for **`ftcomp`** apply. The executable name, taken from `file` and `cerrorsfile`, is set to `ftccers`.

The shell script takes a single option `-g`, and any number of arguments specifying the programs to be compiled. Each argument has `.ftc` appended to give a source name and also specifies an executable program to be produced. The `-g` option, if specified, indicates that the executable program(s) produced should be executed.

The script will provide default value for the following environment variables, if they are not set:

`FTC_FLAGS, LD_LIBRARY_PATH, BINDIR, INCDIR, WBDIR`

To override the default, set the appropriate variable.

Consider the following examples:

- `ftc -g ftprog1` will compile the source `ftprog1.ftc` into the executable `ftprog1` and then execute it.
- `ftc ftprog1 ftprog2` will compile the source `ftprog1.ftc` into the executable `ftprog1` and compile `ftprog2.ftc` into the executable `ftprog2`.

A.2.4.1.3 Sample Scripts

The scripts are `sample.login` for `csh` and `sample.profile` for `ksh` and `sh`. These set the environment variables (described above) appropriate to the machine and directories into which FILETAB has been installed.

A.2.4.2 Libraries

A number of run-time support libraries in `LIBDIR` are supplied with FILETAB for UNIX:

`libftr.so`	The general run-time support library. This should be linked with all FILETAB programs
`libftrcob.so`	The Micro-Focus COBOL support library. This should be linked with FILETAB programs that use `OR=I` files
`libftrauto.so`	Contains the fixed logic support library

A.3 Windows NT Operations Guide

A.3.1 Installation

The steps required to install FILETAB for Windows NT depend on the delivery mechanism. These are:

Delivery mechanism	Format	Steps required
File (Download)	SFX	SFX Installation, see *SFX Installation on page 8*
Media (CDROM)	InstallShield	InstallShield Installation, see *InstallShield Installation on page 8*

A.3.2 SFX Installation

Save the download file to a temporary directory. This file is a self-extracting Wiz file which, when executed, will prompt the user to extract the media InstallShield setup files to the same directory.

When this process is complete proceed with the InstallShield Installation outlined below.

A.3.3 InstallShield Installation

Locate and execute the InstallShield set up application in the file `setup.exe` within the relevant media directory or the temporary directory (SFX Installation). The InstallShield process will guide you through the following:

- Licence agreement dialogue
- Software directory dialogue
- Program group dialogue
- Installed C compiler dialogue
- FILETAB software installation
- Start menu *Filetab for Windows NT* entry

A.3.4 Additional Steps Before Using FILETAB

The standard FILETAB installation sets the working directory to the one selected for installation of the FILETAB software. You may wish to change this by amending the **Start In** option of the **Short Cut** properties window for the *Filetab for Windows NT* entry added in the previous step.

Also note that if the installation was unable to detect the installed C compiler you may need to amend the file `ftc.bat`.

A.3.5 To Remove FILETAB from Your System

You may remove FILETAB from your system using the **Add/Remove Programs** icon within the Windows **Control Panel**. Select the *Filetab for Windows NT* entry from the list of installed software and the **Add/Remove** button. UnInstallShield will guide you through the process of removing FILETAB from the computer.

A.3.6 Directory Structure and Environment Variables

The FILETAB software components will be installed into three subdirectories of a FILETAB root directory. A batch file `ftc.bat` is also installed to aid compilation of FILETAB program source files – this uses the following environment variables:

`FTCDIR`	FILETAB root directory	
`BINDIR`	FILETAB executables	(`%FTCDIR%\BIN`)
`LIBDIR`	FILETAB libraries	(`%FTCDIR%\LIB`)
`INCDIR`	FILETAB include files	(`%FTCDIR%\INCLUDE`)
`CMPDIR`	C compiler directory	
`FTC_COMPILER`	C compiler type	(example: `MSVC` or `GNU`)
`FTC_CC`	C compiler command	(example: `CL` or `GCC`)

A.3.7 ftc Batch File

Command: **`ftc`** *`file`*

This calls the FILETAB executable `ftcomp` to compile the specified FILETAB program source into C source code and then call the relevant C compiler to compile and link the C source code to produce an executable.

The program source and executable file names are taken from the `file` parameter by appending `.ftc` and `.exe` respectively and cerrorsfile is set to ftcerrs.txt.

A.3.7.1 ftcomp Executable

Command: **ftcomp** *`[options] file cerrorsfile`*

The FILETAB compiler (ftcomp) called by the ftc batch file uses the following additional environment variables:

`FTC_FLAGS`	Flags passed to the FILETAB compilation phase
`FTC_CFLAGS`	Flags passed to the C compilation phase
`FTC_LDFLAGS`	Flags passed to the C link phase

The **`ftcomp`** command takes the following arguments:

`file`	FILETAB program source to be compiled
`cerrorsfile`	Errors detected by the C compiler/linker

The `options` can consist of one or more of the following:

`-iPath(s)`	A list of directories, each separated by a colon (:), which will be searched instead of those specified in INCDIR as above.
`-rFile(s)`	A list of initialisation files, each file separated by a colon (:). The first must be the configuration file (see below). Following the configuration file can be a list of FILETAB source files which are evaluated before the specified source is compiled. These can be used, for example, to set ‘default’ options. The default initialisation files are: `ftcconfg.h:.ftcrc`
`-cTargetC`	The intermediate C source (.c) file which is produced by the compiler. Default is `ftctargt.c`
`-hTargetH`	The include (`.h`) file which is produced by the compiler. Default is `ftctargt.h`
`-DFlag`	Defines a conditional compilation flag.
`-oFile`	Specifies the name of the executable file to be produced and also that the C compiler is called to produce it

As well as the two target files above `ftcomp` sends source listings out to `stdout`. Any compilation errors and warning are sent to `stderr`.

A.3.8 Libraries

A number of run-time support libraries in `LIBDIR` are supplied with FILETAB:

`ftr.lib` or `libftr.a`	The general run-time support library
`ftrauto.lib` or `libauto.a`	The fixed logic support library
`ftrodbc.lib` or `libodbc.a`	The ODBC support library

A.3.9 Include Files

`INCDIR` contains the initialisation files, and other include files. These are:

A.3.9.1 The Configuration File

The configuration file (`ftcconfg.h`) is used by the FILETAB compiler (`ftcomp`) and also included by the generated Target C program. It performs the following functions:

(1) Defines the FILETAB Reserved Words

The compiler extracts the Reserved Word definitions, for use by FILETAB programs, from this file. In addition when included in the Target C program it defines the data areas used for Reserved Words.

(2) Defines the 'Code Generation Table'

It also contains a section which defines all the verbs and operations that can be used in DETABS, and specifies the C macros which will be generated from them. This is used by `ftcomp`.

(3) Defines the Initialisation Function

Additionally it contains the C function called by FILETAB at the start of execution to initialise the Reserved Words, buffers and so on.

A.3.9.2 FILETAB Options File

This FILETAB source file (`.ftcrc`) is by default evaluated before the main source is compiled. It allows a site (or user) to specify a (default) set of options which will be applied before every source compiled. This in combination with the configuration file allows a site to configure its copy of FILETAB (for example to make it behave more like UNITAB than FILETAB for VME). This file is used by the FILETAB compiler.

A.3.9.3 The Macros File

The macros file (`ftrmacrs.h`) contains a number of C macro definitions which match the (DETAB) code generated by the FILETAB compiler (from the code generation table) to FILETAB run-time support functions and standard C library functions. This file is included in the Target C program.

A.3.9.4 Type Definition and Portability Files

The following include files that define data types used within the Target C program in a portable manner:

`ftcrtyps.h`	Types used within FILETAB programs
`boolean.h`	The type boolean
`nccport.h`	Portable definitions, with compilation flags for the different machines types supported
`nccstd.h`	Portable definitions, with compilation flags for the different machines types supported

A.3.9.5 Run-time Support Function Prototypes

`rdetsupp.h`	Run-time support functions used within DETABs
`ftr_opts.h`	Encoding of run-time `*OPT`s
`ftcver.h`	Contains the version number (used in MARK)

A.3.9.6 Standard C (Library) Include Files

These are the standard C include files which are also included in the Target C program. They should be supplied with the C development system:

```
malloc.h
select.h
stdio.h
stdlib.h
string.h
time.h
types.h
```

A.3.10 Compiling Programs

Compiling should normally be performed using the `ftc` batch file.

A.3.10.1 Compiler Output

Normally, during compilation, output (other than error reports) appears on `stdout`. Error reports appear on `stderr`. These may be redirected using standard redirection notation:

`ftc src1`	- no redirection
`ftc src1 1> src1.lst`	- `stdout` redirected to file
`ftc src1 1> src1.lst 2>scr1.err`	- `stdout` redirected to file, `stderr` appended to errors file

A.3.11 Running Programs

Programs created using the FILETAB compiler are run from the command prompt by typing the program name followed by the **Enter** key.

A.3.11.1 Input and Output Files

An executable FILETAB program will, by default, identify the names of files either from:

- Environment variables
  where the variable name matches the FILETAB local-name and its value gives the actual file name, or
- Command line arguments
  as follows where executable-name is the name of the program:

```
executable-name [options] main-file [report-file]
```

or

```
executable-name [options]
```

where:

`main-file`	is the actual name of the main (*FILE) file to be accessed by a program which uses Fixed Logic
`report-file`	is the name of the file in which any report will be placed (the print file)
`options`	can be:
	`-v local-name=actualname` Where `local-name` is the FILETAB local name for a file and `actual-name` is the actual (UNIX) name of the file to associated with that local name
	`-o report-name` Also allows the name of the file in which any report will be placed (the print file) to be specified

If the print file name is not specified printed output will be returned to the standard stream `stdout`. DISPLAYs are also sent to `stdout`. Any run-time errors encountered by FILETAB programs are reported to `stderr`.

A.3.11.2 Temporary Sort Files

The FILETAB sort routines attempt to perform their task in memory. If this is not possible then temporary sort workfiles are used. The name of the directory in which the temporary files are to be created is taken as (in decreasing order of precedence):

- The value of the environment variable, `TMPDIR`, in the user's environment
- The `path_prefix` defined as `P_tmpdir` in the `stdio.h` header file
- `/temp`

The temporary files are created with the prefix `SF`.

A.3.12 Result Codes

A.3.12.1 Compile-time Result Codes

`ftcomp` returns a number of result codes.

`COMPIL_OK`	0	no errors, not partprog
`COMPIL_ERRS`	1	errors in compilation

A.3.12.2 Run-time codes

The status of the program is returned as either `EXIT_SUCCESS 0` or `EXIT_FALSE 1`.

Appendix B
Additional Field Types: Dates, Strings, Floating Points

B.1 Date Fields

These are fields which hold binary dates as days since 31/12/1899. This format is used by a number of packages and tools.

B.1.1 Field Definition

Date fields are defined using the basic format:

```
start-position:4<DATE>
```

Any of the alternative field definition formats can be used apart from those containing (`indirect-length`), as the length of a date field must be 4, `start_position` should be word aligned.

B.1.2 Operations and Date Fields

The only operations which can be applied to date fields are:

- Date-Field **+** Field
  The second field will be converted to a binary field if required.
- Date-Field **–** Field
  The second field will be converted to a binary field if required.
- Date-Field `Cmp` Field
  Where `Cmp` is any Comparison operation.

  The second field will be converted to a date field if required

  The second field can be a literal (see date literals below).
- A Move Operation (**`MV`**)
  The second field will be converted to the type of the first field .
- A `SPLITDATE` verb can also be used with a date field.

B.1.3 Conversion To and From Date Fields

- When moving a date field to a binary, unsigned, floating point or any other kind of numeric field the date will be converted into the number of days since 31/12/1899
- When moving a binary, unsigned, floating point or any other kind of numeric field to a date field the numeric field will be treated as containing the number of days since 31/12/1899

- When moving a character field, or literal, to a date field it will be converted if it contains a character date in one of the following formats:
  - dd/mm/yy
  - dd/mm/yyyy
  - ddmonyy
  - dd/mon/yy
  - ddmonyyyy
  - dd/mon/yyyy

  Where:

dd	is the day of the month
mm	is the month number
mon	is the abbreviate month name
yyyy	is the year number including the century
yy	is the year number determined by the current date window, see **OPTION DW on page 4*
/	can be any single character which is not a space or a digit
spaces	will be allowed before the dd, mm, yy, and yyyy

- When moving a date field to a character field it will be converted into a character format specified by the `*OPTION DF`, see **OPTION DF on page 3*. If too many characters are produced they will be truncated to the right, if too few characters are produced they will be padded to the right with spaces

B.1.4 Date Literals

These are specified in single quotes in one of the character formats detailed above. For example:

```
C 0:4<DATE> > '01/08/2000'
  [ Is date later than 1st of August 2000
```

B.1.5 Associating Date Fields with FSCs in the *INLIST (or a *LIST)

A date field can be associated with a Transfer or Control FSC. Only one field should be associated with each FSC. If a date field is associated with a Totalling FSC it will be converted to the number of days since 31/12/1899.

B.1.6 Printing of FSCs Associated with Date Fields

In `*OUT`s and `*HEAD`s the following rules will apply to FSCs mapped onto date fields:

- They will not wrap around, that is:
  - Separate blocks of the same FSC will separately output the same date
  - If an FSC block is too big for the date it will be left justified and padded with spaces
  - If the FSC block is too small the date will be truncated
- The normal editing characters, including single spaces, can be applied to the FSCs. These will be applied to the date already formatted using the `*OPTION DF`

B.1.7 *OPTION DF

This option allows the format of date fields when converted to characters, or printed, to be specified; its format is:

```
*OPT DF=format
```

`format` can consist of any characters apart from space and comma. It can include a number of special format codes, introduced by a % character, which indicate parts of the date to be placed at the corresponding position in the character field or output. The formatting codes are:

Code	Description
`%a`	Abbreviated weekday name
`%A`	Full weekday name
`%b`	Abbreviated month name
`%B`	Full month name
`%d`	Day of the month as a decimal number (01-31)
`%j`	Day of the year as a decimal number (001-366)
`%m`	Month as a decimal number (01-12)
`%U`	Week of the year as a decimal number, with Sunday as the first day of the week (00-51)
`%w`	Weekday as a decimal number (0-6; Sunday is 0)
`%W`	Week of the year as a decimal number, with Monday as the first day of the week (00-51)
`%x`	Date representation for current locale
`%y`	Year without the century as a decimal number (00-99)
`%Y`	Year with the century as a decimal number
`%%`	Percent sign

for example:

```
*OPT DF=%d-%b-%Y
 [ Specifies 1st August 2000 to be output as
 [ 01-Aug-2000
```

The default is `%d/%m/%Y (dd/mm/yyyy - 01/08/2000).`

B.1.8 *OPTION DW

Date windowing has been introduced for use with character dates which have the year held in two digits. When those dates are used (moved to or compared with date fields) the date window setting is used to determine the century. A new `*OPTION DW` is used to specify either a fixed or a sliding century window for determining the century. The default is a sliding date window of:

```
'current year - 80 to current year + 20'
```

(1) Specifying a FIXED century window

```
*OPTION DW=nnnn
```

Use this format to provide a fixed 100 year date window independent of the current year.

`nnnn` is the 4 digit start year of the fixed date window to be used for 2-digit years. For example:

`*OPTION DW=1980` gives 1980-2079 as the date window

2 digit will be taken as 4 digit

```
40 ..................... 2040
60 ..................... 2060
81 ..................... 1981
```

(2) Specifying a SLIDING century window

```
*OPTION DW=mm
```

Use this format to provide a sliding date window dependent on the current year (of program run).

`mm` is number of years ago that the date window starts. For example:

```
*OPTION DW=35
```

run in 1999 gives 1964-1999 AND 2000-2063

run in 2005 gives 1970-1999 AND 2000-2069

run in 2025 gives 1990-1999 AND 2000-2089

The default date window is equivalent to:

```
*OPTION DW=80
```

B.1.9 SPLITDATE Verb

```
SPLITDATE char_field date_field
```

This verb splits a date field `date_field`, which must be `longword` aligned, into a character field `char_field` which should be thirty six

bytes long. The `char_field` will contain information about the date as follows:

char_field+12:4 Day of the month (1-31)

char_field+16:4 Month (0-11; January = 0)

char_field+20:4 Year (current year minus 1900)

char_field+24:4 Day of week (0-6; Sunday = 0)

char_field+28:4 Day of year (0-365; January 1 = 0)

B.2 String Fields

These are fields which hold variable length (`NULL` terminated) strings, such as those held in the environment variables, and handled by the standard library functions.

B.2.1 Field Definition

String Fields are defined using the basic format:

```
start-position:length<STRING>
```

Any of the alternative field definition formats can be used. The length should be 1 more than the maximum number of characters to be held in the field.

B.2.2 Treatment of String Fields

When a String Field is moved to a Character Field:

- If the contents are shorter than the Character Field it is padded with spaces to the right
- If the contents are longer than the Character Field it is truncated to the right
- The `NULL` terminator is removed

When a Character Field is moved to a String Field:

- Trailing spaces are removed
- If the contents are still longer than the String Field it is truncated to the right
- A `NULL` terminator is added

For all other operations on a String Field it is automatically converted to and / or from the corresponding Character Field.

Note: For efficiency improvements, String Fields should be moved to Character Fields (in the same position) before being further manipulated or passed to the fixed logic.

B.3 Floating Point Fields

These are fields which hold floating point numbers, which can be single or double precision.

Note: Numbers held in floating point fields may not have an exact decimal value.

B.3.1 Field Definition

Floating fields are defined using the basic format:

`start-position$length`

Length can be 4 or 8. Any of the alternative field definition formats can be used apart from those containing `indirect-length`.

B.3.2 Operations and Floating Point Fields

The operations which can be applied to Floating Point Fields are:

- Arithmetic Operations
- Comparison Operations
- Move Operation (**MV**)

In each case the second operand is converted to the type of the first operand (see below for details).

- Zeroise Operation

Operations which cannot be applied to Floating Point Fields are:
One-of-a-set, Spacefill, Fill, Logical operations and Multiple Move.

B.3.3 Conversion To and From Floating Point Fields

- When converting a Floating Point Field to a binary, or unsigned field the floating point value will be rounded
- When converting a binary, or unsigned field to a Floating Point Field the result will be a whole number
- For Date Fields see B.1 Date Fields
- When moving a character field, or literal, to a Floating Point Field it will be converted if it contains a numeric constant consisting of a number optionally preceded by a + or - and optionally followed by a . and another number, for example, `-345.67`. Leading spaces are allowed and trailing spaces ignored
- When moving a Floating Point Field to a character field it will be converted into a character format. By default only the whole number part will be output and the sign will be ignored. To specify that the sign is output then set **`*OPT NIS`**. To specify that a number of decimal places are output use **`*OPTION DP`** as below. If too many characters are produced a run-time error will occur (or if

`*OPT NBME` is set the field will be filled with `*s`), if too few characters are produced they will be padded to the left with spaces

B.3.4 Numeric Constants

Numeric Constants when used with Floating Point Fields or other real number type fields can consist of a number optionally preceded by a + or - and optionally followed by a . and another number, for example: `-345.67`

B.3.5 Associating Floating Point Fields with FSCs in the *INLIST (or a *LIST)

A Floating Point field can be associated with a Transfer, Control or Totalling FSC. Only one field should be associated with each FSC.

The default Totalling FSCs are binary fields. To make all totalling FSCs floating point (`$8`) fields then use `*OPT FSCT=5`. To change specific Totalling FSCs to use Floating Fields then use the `*ALTER FSCT5` directive, for example:

```
*ALTER FSCT5[ Change 1 2 and 3 to total using 123[ Floating
Point Fields
```

B.3.6 Printing of FSCs Associated with Floating Point Fields

In `*OUT`s and `*HEAD`s, FSCs mapped onto Floating Point Fields are treated in the same way as FSCs mapped onto binary fields except that the decimal point editing character is used to ‘align’ the value output. That is, the value before any decimal point will be the whole part of the number, and the value after any decimal point will be the fractional part; if no decimal point is specified only the (rounded) whole part of the number will be output.

B.3.7 *OPTION DP

```
*OPT DP=decimal-places
```

Where decimal-places specifies the number of decimal places to be output when moving a Floating Point field to a character field.

The default is 0.

B.4 Other Types of Field

Other Field Types may be allowed in your particular software environment. See *Appendix E Informix C-ISAM* for those allowed with C-ISAM, and the *Appendix G Informix Guide* for those allowed with Informix SQL.

Appendix C
Miscellaneous Features: Conditional Compilation

C.1 Conditional Compilation (*COND)

Conditional compilation can be used to specify that parts of a program are only compiled under certain circumstances, and so is useful for tailoring programs whilst maintaining a single source. The parts of a program which are compiled and controlled by ‘compilation flags’. These can be defined by the **ftcomp** command. The **ftc** command uses an environment variable FTC_FLAGS to define compilation flags. The conditional compilation directives are:

C.1.1 *COND DEFINE Flag

This defines a compilation flag. Flag should be a valid FILETAB name which does not contain -s.

C.1.2 *COND IF Flag

The source after a ***COND IF** (and up to a ***COND ENDALL**, matching ***COND ENDIF** or matching ***COND ELSE**) is only compiled if flag has been defined. ***COND IF**s can be nested.

C.1.3 *COND IFNOT Flag

The source after a ***COND IFNOT** (and up to a ***COND ENDALL**, matching ***COND ENDIF** or matching ***COND ELSE**) is only compiled if flag has not been defined. ***COND IFNOT**s (and ***COND IF**s) can be nested.

C.1.4 *COND ELSE

***COND ELSE** reverses the effect of a matching ***COND IF** or ***COND IFNOT**, for if the source was being compiled it will stop being compiled, or if it was not being compiled it will now be compiled (up to a ***COND ENDALL**, or matching ***COND ENDIF**).

C.1.5 *COND ENDIF

Ends the effect of a matching ***COND IF**, ***COND IFNOT** or ***COND ELSE**.

C.1.6 *COND ENDALL

Ends the effects of all **`*COND IF`**s, **`*COND IFNOT`**s and **`*COND ELSE`**s. Compilation is resumed (unconditionally).

Appendix D
FILETAB on the Web

D.1 Overview

D.1.1 What is FILETAB on the Web?

FILETAB on the Web is the term used to refer to the FILETAB extensions, introduced in Version 3.0, that allow you to access your company data on web browsers.

Your company data may be deployed as:

- *Static* reports
  Pre-generated reports, regenerated at fixed time intervals.
- *Dynamic* reports
  Regenerated by user request; FILETAB programs are run from, and return up-to-date information to, the browser.

D.1.2 Filetab on the Web Features

- Produce HTML output from FILETAB programs
- Update 'index' pages with links to that output
- Add graphics (for example: company logo) to your output
- Access data entered into an HTML form from within FILETAB programs
- Produce default HTML forms automatically during compilation
- Run programs from a browser

D.1.3 What Your Operations Guide Contains

Here we describe those features of the FILETAB that deal with web deployment. It is assumed that you are familiar with the FILETAB language. Please refer to the main chapters in this guide for a general description of the language and to *Appendix A Operations Guide* for information on installation or compilation.

An example program, see *Chapter 11 Examples* will be converted, in stages, for web use.

FILETAB on the Web is divided into the following sections:

D.1 Overview	Describes what is available in this document and how it relates to the FILETAB software
D.2 Adding the Web Features	Simple text output, HTML output, an index page, a company logo or graphics together with how to run a program using HTML or a hypertext link
D.3 More About the New *OPTIONs and FTC_HTML	Explains HTML at run-time and compile-time and gives a summary of compile-time and run-time options
D.4 More About *MAPs	Explains the use of `*MAPs` in connection with CGI, with examples
D.5 Summary	Describes the use of FILETAB keywords in `*MAP` detail lines, with examples
D.6 Examples	Explains use of different input types
D.7 Listing of mapdetail.ftc	Listing of program `mapdetail.ftc`

D.2 Adding the Web Features

To introduce the features we will convert the `test6` program, shown in *Figure D-1: Program Source Code*, for web use.

There are four steps:

(1) Look at the original program: simple text output.

(2) Add HTML output.

(3) Add an index page.

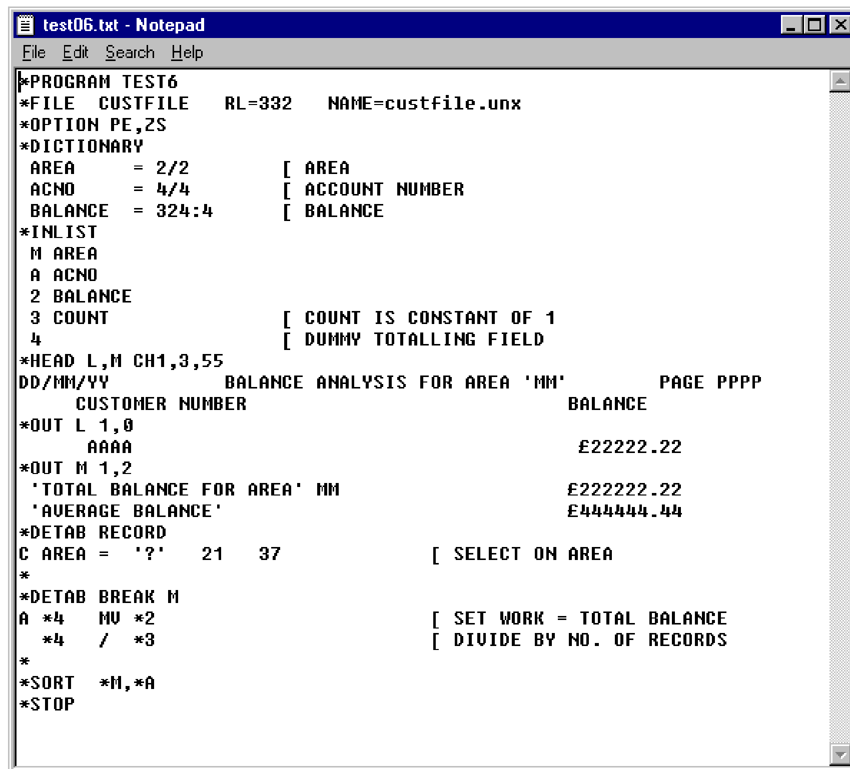
(4) Add a company logo or graphics to the output.

Additionally there are two methods of running the program from a browser:

(1) Run programs using HTML form input.

(2) Run programs using a hypertext link.

D.2.1 The Original Program[1]: Simple Text Output



test06.txt - Notepad

File Edit Search Help

```
*PROGRAM TEST6
*FILE  CUSTFILE   RL=332   NAME=custfile.unx
*OPTION PE,ZS
*DICTIONARY
 AREA     = 2/2        [ AREA
 ACNO     = 4/4        [ ACCOUNT NUMBER
 BALANCE  = 324:4      [ BALANCE
*INLIST
 M AREA
 A ACNO
 2 BALANCE
 3 COUNT               [ COUNT IS CONSTANT OF 1
 4                     [ DUMMY TOTALLING FIELD
*HEAD L,M CH1,3,55
DD/MM/YY          BALANCE ANALYSIS FOR AREA 'MM'        PAGE PPPP
     CUSTOMER NUMBER                            BALANCE
*OUT L 1,0
      AAAA                                       £22222.22
*OUT M 1,2
 'TOTAL BALANCE FOR AREA' MM                    £222222.22
 'AVERAGE BALANCE'                              £444444.44
*DETAB RECORD
C AREA =  '?'   21   37             [ SELECT ON AREA
*
*DETAB BREAK M
A *4   MV *2                        [ SET WORK = TOTAL BALANCE
  *4   /  *3                        [ DIVIDE BY NO. OF RECORDS
*
*SORT  *M,*A
*STOP
```

Figure D-1: Program Source Code

The program is compiled in the normal way:

```
ftc test6
```

Running the program without specifying an output file[2]:

```
test6
```

Produces the output shown in *Figure D-2: Program Output, Non-HTML* which, by default, is sent to the screen.

1. A description of this program in the FILETAB language appears in *Chapter 11*.
2. See *Compiling Programs on page 12 of Appendix A*.

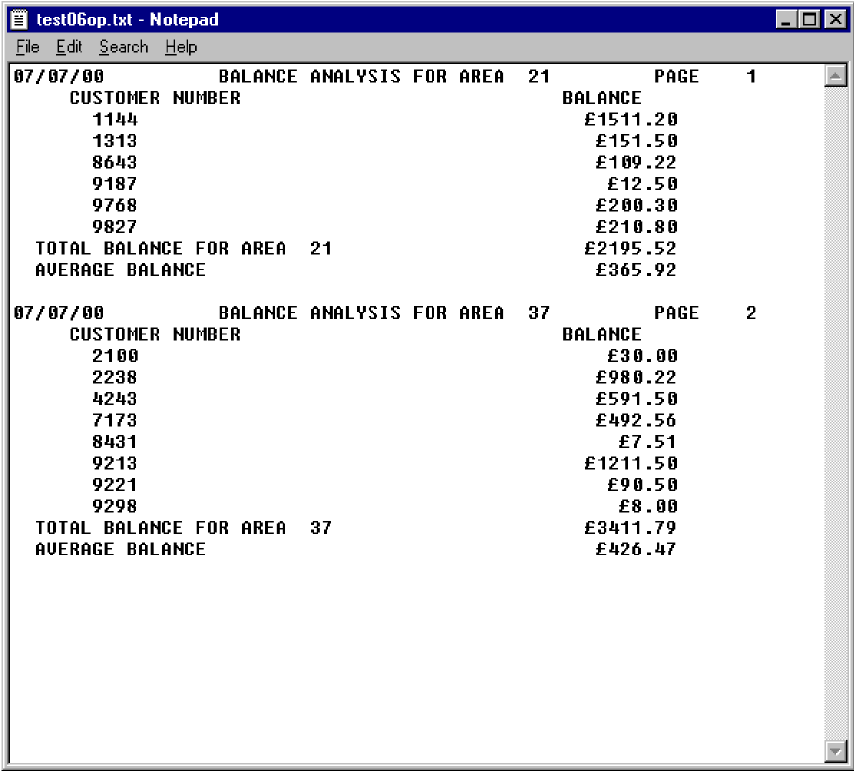


Figure D-2: Program Output, Non-HTML

D.2.2 Add HTML Output

The simple text output in *Figure D-2: Program Output, Non-HTML* could be captured and edited, using a text or HTML editor, for web use. However, FILETAB can produce HTML automatically in two ways. HTML output, and its format, may be specified at: ***run-time*** using environment variables or ***compile-time*** using a series of `*OPTIONs` and reserved words. These topics are covered in a later section.

For now we'll add just one line to the FILETAB source code to produce HTML output:

```
*PROGRAM TEST6
*FILE CUSTFILE          RL=332      NAME=custfile.txt

*OPT HTML                           [add this line to produce HTML
                                    output

*OPTION PE,ZS

*DICTIONARY
                    .
                    .
                    .
```

Figure D-3: Use of *OPTION HTML

Recompile and run the program, this time specifying an output file: `test6 -o t6op.html`.

Adding this line causes FILETAB to prepend and append the appropriate HTML tags, as shown in *Figure D-4: First HTML Output (text editor)*:

```
<HTML>

<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<TITLE>test06.ftc</TITLE>

</HEAD>
<BODY>
<PRE>
   .
   .
[ previous program output appears here...]
   .
   .
</PRE>
</BODY>
</HTML>
```

Figure D-4: First HTML Output (text editor)

We can also change the text within the `<TITLE>` tags by using the FILETAB `*TITLE` directive:

```
TITLE
Filetab on the web output
```

Once these lines have been added, your output should be similar to that shown in *Figure D-5: Our First HTML Output (browser).*

Figure D-5: Our First HTML Output (browser)

D.2.3 Add an Index Page

You can browse a single report by specifying its file path in the browser. However, as the number of reports grow, you may want to construct an index page using your preferred text editor containing a menu of hypertext

links so that the different reports are accessible with a single click of the mouse.

```
<HTML>
<HEAD>
<TITLE>Simple index page</TITLE>

</HEAD>
<BODY>

<B>My reports...</B>
<BR><BR>
<A HREF="t6op.html">Areas 21 and 37  (test6 program output)</A>

</BODY>
</HTML>
```

Figure D-6: A Simple Index Page (text editor)

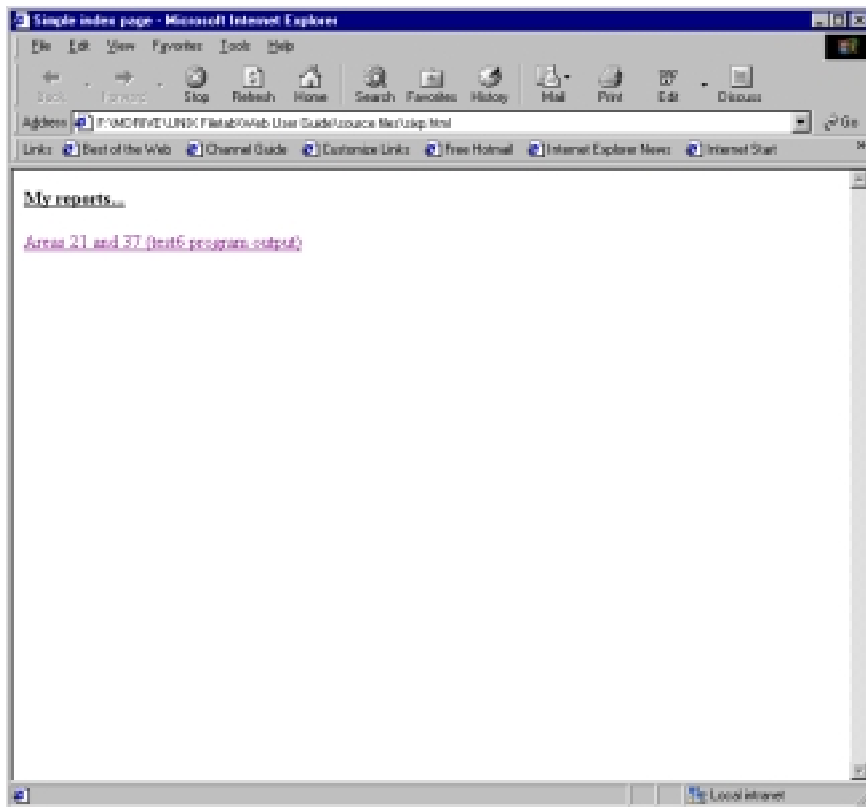


Figure D-7: A Simple Index Page (browser)

Clicking the link:
`Areas 21 and 37 (test6 program output)`
will deliver `t6op.html` to the browser.

D.2.3.1 How to Add Indexing

The entries in the index pages can be created manually, however, *FILETAB on the Web* can add them automatically when the program is run. This is illustrated in the *Figure D-8: Automatic Entries:*

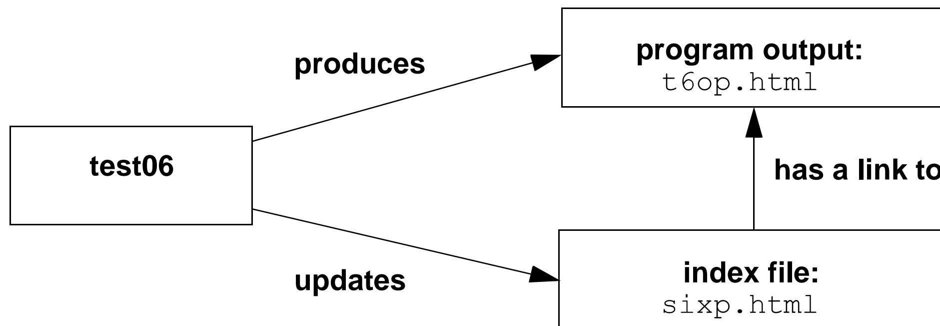


Figure D-8: Automatic Entries

To make FILETAB add indexing to the HTML output, you need to:

(1) Switch on indexing.

(2) Tell FILETAB the name of the index file.

(3) Tell FILETAB the text to appear as the link.

(4) Tell FILETAB where to put the link.

These four steps use `*OPTIONs` and reserved words and are described in detail below:

(1) Switch on indexing by adding `*OPT HTMLI` at the start of your program.

This option will cause FILETAB to allocate areas of memory for the reserved words it uses for indexing. These reserved words should be given the appropriate values at `*DETAB START`, as shown below.

(2) Tell FILETAB the name of the index file.

```
_HTML-IndexFile  MV  'sixp.html'
```

You will need to create an index file with this name. Sample files are provided with the release in the `examples` directory. Make a copy and then edit them to reflect your requirements (adding your own text, graphics and so on).

(3) Tell FILETAB the text to appear as the link.

```
_HTML-IndexKey/15 MV  'Areas 21 and 37'
_HTML-IndexKey+15/23 MV '(test6 program
output)'
```

Another reserved word `_HTML-IndexInfo` could be used to add text after the link on the same line, that is, if we wanted `Areas 21 and 37` to be the link and (`test6 program output`) to appear after it (as in *Figure D-9: Using _HTML-IndexInfo Reserved Word*) we would add the following code to `*DETAB START`:

```
_HTML-IndexKey  MV 'Areas 21 and 37'
_HTML-IndexInfo MV '(test6 program output)'
```

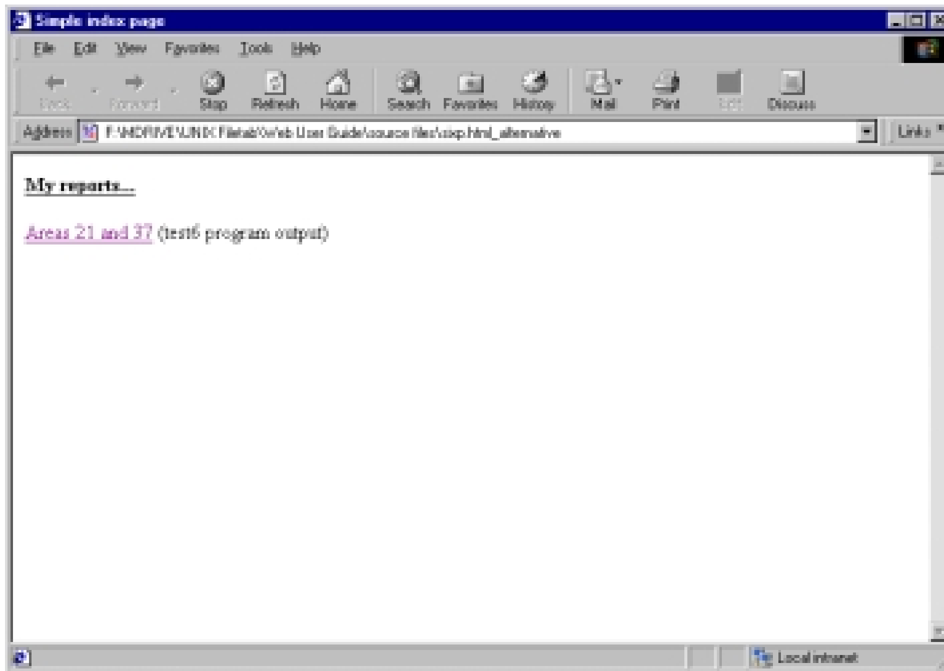


Figure D-9: Using _HTML-IndexInfo Reserved Word

(4) Tell FILETAB where to put the link.

```
_HTML-IndexAfter  MV  'My reports...'
```

By default, the link will be positioned immediately after the `<BODY>` tag in the index file (that is, at the start of the page). Here we have specified that we want the link after the line that contains the text
`My reports....`

D.2.4 Add a Company Logo or Graphics to the Output

The simple HTML output is purely functional - it contains only the data requested. However, we can easily add a company logo or graphics or more complex formatting to the output. The method used to accomplish this is the `dummy *MAP`[1]. The `dummy *MAP` works as follows:

The first `*MAP`, that the FILETAB compiler encounters in a program source, is used to specify two files that are to be output before and after

1. `*MAP`s are described in *Section D4*.

the program output. These files (the header and footer files) contain any HTML required to format the FILETAB output.

The filenames are specified using the HEADER and FOOTER keywords on the *MAP directive line, for example:

```
*MAP  dummy HEADER=stdouthdr.html,FOOTER=stdoutftr.html
```

- The name of the map need not be dummy but is used here to reflect the MAP's purpose
- The *OPT FRM option should not appear before this MAP - we do not want to create a form for this MAP

When run, the program contents will be preceded by the contents of the file stdouthdr.html and followed by the contents of the file stdoutftr.html.

The header and footer files can be as simple as:

```
<BODY BGCOLOR="#FFFFFF" TEXT="#000000"> [Set
background
[and
[text colours
```

in the header, and, in the footer

```
</BODY> [HTML tag for end of body
```
[1]

The header and footer will usually be more complex than this and may include any HTML tags.

For example, here we are using tables to format the output giving a side-bar on the LHS of the main output.

```
<BODY BGCOLOR="#FFFFFF" TEXT="#000000">
<TABLE BORDER="2" CELLPADDING="2">
            <TR
            >
                  <TD ALIGN=LEFT VALIGN=TOP WIDTH=100>
                       <CENTER>
                       <IMG SRC="logo.gif" BORDER="0">
                            <BR>- OR -<BR><BR>links<BR> to other pages
                       <BR><BR>- or -<BR><BR>key information
                       <BR><BR>- or -<BR><BR>...
                       </CENTER>
                  </TD>
                  <TD ALIGN="LEFT" VALIGN="TOP">
<PRE>
```

Figure D-10: Sample Header File stdouthdr.html

1. There is no need to have a footer file if this is all it would contain. FILETAB will automatically close the BODY section.

Note: If a header file is present the `<PRE>` *tag is suppressed when FILETAB produces its output (otherwise any formatting introduced in the header would be overridden). Because we still require the tag we must add it to the header file. We also add a closing* `</PRE>` *tag to the footer.*

The footer closes any open tags:

```
</PRE>
</TD>
</TR>
</TABLE>
</BODY>
```

Figure D-11: Sample Footer File stdoutftr.html

Using these files the following output is produced:

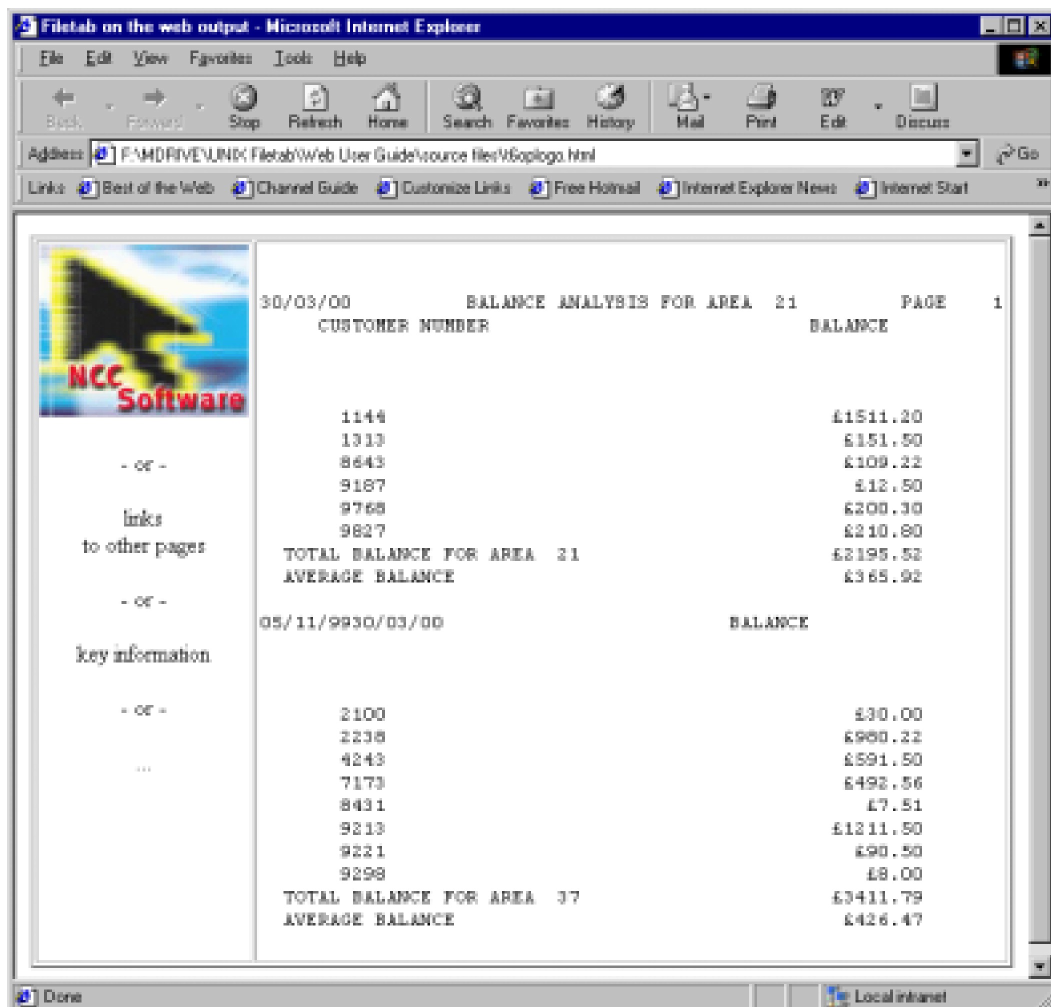


Figure D-12: Program Output with Company Logo

D.2.5 Run the Program Using HTML Form Input

FILETAB on the Web allows you to use the sophisticated input methods available to HTML forms[1] to input data into your programs.

To do this you must:

(1) Tell FILETAB that the program is using CGI.

(2) Tell FILETAB that you are going to read from a form.

(3) Tell FILETAB which program variables are to receive the values read from the form.

(4) Tell FILETAB to read from the form at the start of the program.

You may also:

(5) Tell FILETAB to create a form for you during compilation. (You must also tell it the name of the program.)

D.2.5.1 Form Input: How It's Done

We can show how form input is done by amending the original program. Instead of producing output for areas 21 and 37, we can look at any area by entering the area number as text into a form.

This solution is not ideal[1] but is for illustration only.

(1) Tell FILETAB that the program is using CGI (use of `*OPTION HTMLC`).

The output from the program must have a header of a particular structure so that the browser can understand it. We tell FILETAB to add this header by the use of `*OPTION HTMLC`:

```
*OPT HTMLC
```

(2) Tell FILETAB that you are going to read from a form.

We use a standard `*IFILE` directive in which the file organisation is set to `CGI`. We must also specify the name using a `*MAP` directive which will contain detail lines that map form data onto the program's `DICTIONARY` area. Data can then be transferred to the correct variables when we read the form.

```
*IFILE ARGS  OR=CGI   MAP=mapname
```

`ARGS` is the logical name of the file.

(3) Tell FILETAB the name of the program which will read the form

When a form is submitted, a request is sent by the browser to the server containing, amongst other items, the `ACTION` to be taken. We set this `ACTION` in the `*MAP` directive:

```
*MAP mapname ACTION=http\://yourserver/cgi-
bin/test6
```

The program `test6` residing in the `cgi-bin` directory on the server called `yourserver` will be run when the form is submitted. The output of `test6` will be sent to the browser.

1. These forms use CGI (Common Gateway Interface) which provides a standard mechanism for communication between a Client (running a browser) and a Server machine. The Server usually has a designated directory (commonly called cgi-bin) that contains programs it will attempt to run. The *output* from these programs is sent to the browser.

Note: The additional backslash character (\) that appears before the colon character (:). This is to 'escape' the colon character which would otherwise be misinterpreted by the compiler.

(4) Tell FILETAB which program variables are to receive the values read from the form.

This information is contained in the `*MAP` detail lines:

For example, if we have a `DICTIONARY` item
`'AREAREQ    1000/2'`
we map a form field onto it using the detail line:

```
TYPE=text,NAME=area,PROMPT=Input area\:
:AREAREQ
```

This line may be read as Map the text field called `area` in the form onto the FILETAB `DICTIONARY` area allocated to the variable `AREAREQ'`. The `PROMPT` keyword is for text to be used as a prompt if a default form is produced. Other TYPEs of input and their keywords will be described later.

(5) Tell FILETAB to 'read' from the form at the start of the program (for instance `*DETAB START`).

```
READ ARGS 0/0
```

The effect of this statement is to access the CGI form data and, using the `MAP` information, transfer the data into the appropriate areas of the `DICTIONARY`. The `0/0` is present to fit in with the normal file read statement and is ignored. Each transfer acts like an ordinary `MV` statement that is, data conversion occurs.

(6) Tell FILETAB to create a form for you during compilation (optional).

Using an `*OPT FRM` before a `*MAP` directive will result in a form being automatically created during compilation.This form may be edited as required to reformat text and/or add graphics.

D.2.5.2 The Program Altered for CGI Use: test6.ftc

```
*PROGRAM TEST6
*FILE  CUSTFILE   RL=332   NAME=custfile.txt
*OPTION PE,ZS
*OPTION HTMLC                                          [   1)
*IFILE ARGS OR=CGI MAP=t6args                          [   2)
*DICTIONARY
 AREA     = 2/2        [ AREA
 ACNO     = 4/4        [ ACCOUNT NUMBER
 BALANCE  = 324:4      [ BALANCE

 AREAREQ  = 1000/2     [ AREA REQUIRED defined by CGI form
'read'
```

1. Entering an invalid area number is not flagged as an error (solution: check the value input after the READ or use selection from valid values rather than a free text entry. Other types of input are dealt with later in this guide).

```
*OPT FRM                                                 [  5)
*MAP t6args ACTION=http\://yourserver/cgi-bin/test6      [  5)
 TYPE=text,NAME=area,PROMPT=Input area\:  :AREAREQ       [  3)

*INLIST
 M AREA
 A ACNO
 2 BALANCE
 3 COUNT                [ COUNT IS CONSTANT OF 1
 4                      [ DUMMY TOTALLING FIELD

*TITLE
Filetab on the web output
*HEAD L,M CH1,3,55
DD/MM/YY         BALANCE ANALYSIS FOR AREA 'MM'       PAGE PPPP

     CUSTOMER NUMBER                           BALANCE
*OUT L 1,0
       AAAA                                     £22222.22
*OUT M 1,2
 'TOTAL BALANCE FOR AREA' MM                   £222222.22

 'AVERAGE BALANCE'               £444444.44

*DETAB START
I READ ARGS 0/0                                        [ 4)

*DETAB RECORD
* this line was:  C AREA =  '?'   21   37
C AREA =  AREAREQ   Y           [ SELECT ON AREA REQUIRED

*DETAB BREAK M
A *4   MV *2                    [ SET WORK = TOTAL BALANCE
  *4   /  *3                    [ DIVIDE BY NO. OF RECORDS
*
*SORT  *M,*A
*STOP
```

Figure D-13: Program Listing of test6.ftc

Added/changed lines are indicated by 1) - 5).

We have used the FRM *OPTION so, when test6.ftc is compiled, a default form will be created in a file called t6argsfrm:

Figure D-14: Default Form Generated from MAP

As there is one input field, the form produced does not contain a submit button by default. The form may be submitted by pressing the Return key.

More complex forms (containing titles, headings, prompts, buttons and graphics) may be produced by adding detail lines to the MAP. These lines are discussed later in this guide.

D.2.6 Run the Program Using a Hypertext Link

To illustrate this method we'll amend the index program in *Add an Index Page on page 6* to use the reserved word _HTML-IndexInfo. This will output a link in the index file which will be used to re-run the program.

For example:

Figure D-15: The Added Link to Re-run the Program

`Areas 21 and 37(test6 program output)` is a link to the output (created at the time specified). `Re-run` is a link to the `test6` executable in the `cgi-bin` directory of `yourserver`. When selected this will cause the program to return the output of `test6` to this browser window. This index file is also updated by the program with the date of the new run.

To produce this effect you must take two steps (in addition to the steps in *Add an Index Page on page 6*):

(1) Tell FILETAB that we need to output a special header for CGI.

As in *Form Input: How It's Done on page 12, Step 1*, this is done using `*OPT HTMLC`.

(2) Tell FILETAB to output the link to re-run the program.

We need to define `_HTML-IndexInfo` at `*DETAB START`. This variable is used to hold additional text to appear after the link. In the simplest case we could leave out the date information and make the following assignment:

```
_HTML-IndexInfo+34/45        M    '<A HREF="/cgi-bin/test6"> Re-run</A>'
                             V
```

for a more useful index link as in *Figure D-15: The Added Link to Re-run the Program,* we will build up the message as follows:

```
_HTML-IndexInfo/80     MV   ' - Run on '
_HTML-IndexInfo+11/9   MV   DATE
_HTML-IndexInfo+20/4   MV   ' at '
_HTML-IndexInfo+24/10  MV   TIME
_HTML-IndexInfo+34/45  MV   '<A HREF="/cgi-bin/test6"> Re-run</A>'
```

`DATE` and `TIME` are FILETAB variables which contain the date and time of the program run.

That's it. We've converted the `test6` program for web use. In the following sections we will discuss `*OPTIONs` and `*MAPs` in more detail.

D.3 More About the New *OPTIONs and FTC_HTML

FILETAB can produce HTML automatically in two ways. HTML output, and its format, may be specified at:

- ***Run-time*** using environment variables or
- ***Compile-time*** using a series of `*OPTIONs` and reserved words

D.3.1 Switch on HTML Output at Run-time

With this method, no changes need be made to the FILETAB programs. The switch to `HTML` output is made using the `FTC_HTML` environment variable. This is set to a comma-separated list of options indicating the required output format and page handling.

There are three main types of option (summarised in *Figure D-16: The Use of *OPTIONS and the FTC_HTML Environment Variable*):

- Formatting option: `N`
- Page handling options: `F` and `P` (these are mutually exclusive)
- Miscellaneous options: `C`, `I` and `E` (`I` and `E` are mutually exclusive)

D.3.2 Switch on HTML Output at Compile-time

With this method, changes to the FILETAB source are required. The switch to HTML output is made using a series of `*OPTIONs` each having an effect equivalent to a run-time option, described in *Switch on HTML Output at Run-time on page 17,* with one additional `*OPTION`:

```
*OPT NHTML
```

This option prevents any run-time changes to the HTML options (other than setting the indexing reserved words within the FILETAB program). Used on its own it prevents HTML output from being produced.

D.3.3 Summary of Compile-time and Run-time Options

The effects of the compile-time and run-time options have been summarised

Compile-time	Run-time							HTML Output ?	Effect
*OPTION	FTC_HTML set?	Contains							
		N	F	P	C	I	E		
NHTML	No	-	-	-	-	-	-	No	Ordinary program output
HTML	Yes	N	-	-	-	-	-	Yes	<PRE> pre-formatted program output. </PRE>
HTMLN	Yes	Y	-	-	-	-	-	Yes	Uses a fixed-pitch (Tele Type) font and nbsp's to format the output
-	Yes	-	N	N	-	-	-	Yes	No page handling (output is continuous in one file / web page with no page breaks)
HTMLF	Yes	-	Y	N	-	-	-	Yes	One file (web page) per page of output, with First, Next, Previous and Last page links
HTMLP	Yes	-	N	Y	-	-	-	Yes	All output in one file (web page) with First, Next, Previous and Last page links
HTMLC	Yes	-	-	-	Y	-	-	Yes	Output HTTP content type before HTML (for use as a CGI response)
-	Yes	-	-	-	-	N	N	Yes	No index handling
HTMLI	Yes	-	-	-	-	Y	N	Yes	Index handling required. Allocate areas for the indexing Reserved Words
HTMLE	Yes	-	-	-	-	N	Y	Yes	Index handling required. Don't allocate areas for indexing Reserved Words

Figure D-16: The Use of *OPTIONS and the FTC_HTML Environment Variable

Shaded areas show mutually exclusive options. For these, only the combinations shown are permissible.

D.4 More About *MAPs

MAPs are mainly used to tell FILETAB where to put the data it reads (they *map* the input data from the record area on to the `DICTIONARY` area). They may also tell FILETAB about indicator variables that may be tested to check that input has been received.

If the `*OPT FRM` option has been exercised then the `MAP` may also contain information used to generate default input forms.

D.4.1 Using *MAP Detail Lines to Read from CGI Forms

D.4.1.1 File Organisation OR=CGI

To treat the CGI form as an input file, we need a `*IFILE` directive in the program which tells FILETAB that its organisation is CGI.

```
*IFILE logical_name   OR=CGI   MAP=mapname
```

When FILETAB reads from this file, it will need to know where to put the data it reads. This is done using the MAP named in the `*IFILE` directive.

D.4.1.2 MAP Format for CGI

The format of the `MAP` directive and detail lines for use with CGI organisation is:

```
*MAP mapname [titleactionmethod]
[fieldattributes] :fieldname [:indicator]
```

where:

`titleactionmethod` Is a list of comma-separated keyword-value pairs to specify:

- (`KEYWORD` is `TITLE`) the title of the form (web page) `created. The default value` is `Input Form`
- (`KEYWORD` is `ACTION`) the action the form is to take when the form is submitted. Usually the `URL` for the FILETAB program to be run. *The default value is* "" (no action specified)

Note: If the value associated with the `ACTION` *keyword begins with an '*`$'` *character.* *FILETAB expects an environment variable name to follow it. This environment variable should then hold the required URL.*

- (`KEYWORD` is `METHOD`) the method by which arguments are passed to the program. *The default value is* `POST`

`titleactionmethod` only takes effect if `*OPT FRM` is exercised

`mapname` Is the name of the MAP (`<mapname>frm.html` will be created for this MAP if `*OPT FRM` is in effect)

`fieldattributes` Is a list of comma-separated keyword-value pairs to specify:

- (`KEYWORD` is `NAME`) the name of the field (where applicable)
- (`KEYWORD` is `TYPE`) the type of input field (text, textarea, password, checkbutton, radiobutton, listbox, and reset or submit button). Also used for specifying non-input literal text and/or images
- (various `KEYWORDs`) prompts, sizes and default values for input fields/buttons (where applicable)
- (`KEYWORD` is `FORMAT`) complex formatting
- `fieldattributes` keyword-value pairs are summarised in *Summary on page 22*

`fieldname` Is the name of the target field (required). For input items (not literal text, images or formatting) this name must appear in a FILETAB `*DICTIONARY`.

Note: `Fieldname` *is required for non-input items even though there is no target FILETAB variable. In these cases, the actual name specified should NOT appear in a* `*DICTIONARY` *but may be used to comment the line (for example* `:image` *or* `:format`*)*

`indicator` Is the name an indicator variable associated with the target field (optional). The indicator variable should be a `:2` field. It is used to indicate whether data has transferred to the target field from the form (`<0` if no transfer)

D.4.1.2.1 MAP Directive Examples

```
*MAP if1 TITLE=My title,ACTION=http\://yourserver/cgi-bin/
test6
```

Assuming `*OPT FRM` is in effect this may be read as: Create a form called `if1frm.html` with the title of `My title` which, when submitted, will run the program `test6` using the `POST` method.

```
*MAP if2 ACTION=http\://yourserver/cgi-bin/test6
```

Create a form called `if2frm.html` with the title of `Input Form` (the default) which, when submitted, will run the program `test6` using the `POST` method.

```
*MAP if3 METHOD=GET,ACTION=http\://yourserver/cgi-bin/test6
```

Create a form called `if3frm.html` with the title of `Input Form` (the default) which, when submitted will run the program `test6` using the `GET` method.

```
*MAP if4 ACTION=$IF4ACTION
```

Create a form called `if4frm.html` with the title of `Input Form` (the default). Get the `ACTION` from the `IF4ACTION` environment variable (at compile-time). Use the `POST` method (default).

```
*MAP if5
```

Create a form called `if5frm.html` with the title of `Input Form` (the default). `ACTION=""` which will produce a run-time message from your server.

D.4.1.2.2 MAP Detail Examples

These examples have been chosen to show the effects of a number of keyword-value combinations for the default input `TYPE` (text). They illustrate how the keyword-value notation is used.

For these examples assume that `AREAREQ` has been defined in a FILETAB `DICTIONARY`.

```
NAME=area        :AREAREQ
```

Create a text input box (the default if `NAME` is specified) of default size.

Note: No prompt, no submit button.

```
NAME=area,PROMPT=Enter area number       :AREAREQ
```

Enter area number

|

Note: We have a prompt.

```
NAME=area,PROMPT=Area,VALUE=21           :AREAREQ
```

Area

21

Note: We have a default value.

```
NAME=area,PROMPT=Area,SIZE=2,MAXLEN=2    :AREAREQ
Area
```

Note: Only two characters allowed.

TYPE=password,NAME=area,PROMPT=Area (secret),SIZE=2,MAXLEN=2
:AREAREQ

Area (secret)

**

Note: We have changed the `TYPE` to password, now asterisks are displayed instead of actual characters.

D.4.1.3 Doing the READ

The file is read by issuing a `READ` statement at, for instance, `DETAB START`.

```
*DETAB START
I READ logical_name  0/0
```

The second operand of the `READ` verb is not used as the `MAP` says which fields are to be defined.

The read transfers data to the FILETAB DICTIONARY area.

D.4.2 Using *MAP Detail Lines to Create Better Input Forms

The input form we created is functional but can be improved by adding, for example, a heading, a submit button and perhaps some explanatory text. Graphics may also be added to improve the appearance of the form (or make the form 'fit in' with other company documentation).

In addition, we are not limited to input of TYPE=text:

text	simple text box (default if NAME keyword is present).

Other input TYPEs include:

password	where the text typed is automatically replaced by asterisks.
radio/ checkbox	where a mouse is used to select or deselect an item.
select	where items are chosen from a pull-down menu.
textarea	where text may be typed into a multi-line entry box.
submit	places a submit button on the form - to send the form for processing.
reset	places a reset button on the form - to reset values to their defaults.

Other settings of TYPE are used, not for input, but to cause the inclusion of text and/or images and for creating sophisticated layouts in the form:

literal	used to place literal text into your input form. *Note:* Literal *is the default if there is no* NAME *keyword.*
image	used to include a graphic in the form.
format	used to exert more control over the layout of your form.

We have seen the NAME and PROMPT keywords used with TYPE=text. Each TYPE above has a number of keywords associated with it, these specify any values that are required by that type of input or modify its operation. FILETAB will, in some cases, provide defaults if a keyword-value pair is absent. Keyword-value pairs are summarised below.

D.5 Summary

This section describes how to use FILETAB keywords in *MAP detail lines. If *OPT FRM is in effect, the FILETAB compiler converts the information in the detail lines into HTML tags in an input form.

Inputs (as well as text and some formatting) are specified using the TYPE keyword.

For example:

TYPE=text   for a text box input of default size.

with other keyword-value pairs, appropriate to the input TYPE, being used to further control the input.

For example:

TYPE=text, SIZE=23   for a text box input of particular size.

TYPE keywords fall into three categories:

- TEXT INPUT, TEXT OUTPUT (literals) and IMAGES
  Keywords: text, password, textarea, literal, image
- BUTTON SELECTED INPUT
  Keywords: checkbox, radio, select, submit, reset
- FORMATTING
  Keywords: format

These TYPEs and associated keywords are summarised in the following tables. Examples using the different TYPEs follow the tables.

	FILETAB Keywords	
input_type values	**NAM*E** a	**VAL*UE** a
format	Not required	Use for specifying formatting for *following* MAP detail lines.

Figure D-17: Keywords for FORM Generation: Formatting

HTML tags are generated from the FILETAB keywords. Input types are specified using the TYPE keyword and the input_type values.

D.6 Examples

Examples of the use of the different TYPEs have been amalgamated into the file mapdetail.ftc installed in the $FTC_ROOT/examples/Web directory. Compilation of this file will produce a number of input forms. To facilitate easy browsing, one of the forms, topfrm.html, contains links to the others.

Each TYPE in the list below is the subject of one or more MAPs in the program. To make cross-referencing easier, the map names have been based on the list numbering.

TYPE=text	MAP ia	Create a text input box (default TYPE when NAME specified) of default size. *Note: Only the text box is visible with no prompt or submit button. A submit button is not required if there is a single input field.*
	MAP ib	Create four text boxes. Use of PROMPT, VALUE, SIZE and MAXLEN. *Note: A submit button has been automatically added. Poor layout of input boxes due to unbroken lines. Use of default* LITERAL *no* TYPE *or* NAME *makes* LITERAL *the default).*
	MAP ic	As for ib but with line breaks added (two methods).
TYPE=password	MAP ii	Asterisks are displayed when text is entered.
TYPE=textarea	MAP iii	Create three textareas. Use of SIZE and VALUE keywords.
TYPE=literal	no MAP	*No* MAP *to display.*
TYPE=image	MAP v	Including images in a form, making one a link.
TYPE=checkbox	MAP vi	Create three checkboxes selections (using PROMPT). Use of CHECKED keyword.
TYPE=radio	MAP vii	Create four radio buttons selections (using PROMPT). Use of CHECKED keyword.
TYPE=select	MAP viii	Create three selections (using PROMPT). Use of SIZE and SELECTED keywords.
TYPE=submit	MAP ix	Change the text on the button using VALUE keyword.
TYPE=reset	MAP x	Create a button to reset the fields to their original values. Change the text on the button using VALUE keyword.
TYPE=format	MAP xia	Attempt to create a complex form (no formatting). *Note: Poor layout of input items.*
	MAP xib	Add formatting to xia. Change the formatting part way through a form.
	MAP xic	Additional formatting techniques. Use *COPY files in formatting to reduce typing and increase readability.

D.7 Listing of mapdetail.ftc

```
*
**************************************************************
******
*PROGRAM MAPDETAIL
* ****************
* This program is for COMPILATION ONLY to demonstrate
creation of input
* forms by the FILETAB compiler.

* *OPT FRM is used to create forms. *OPT NFRM switches this
off so:
* *OPT FRM
* *MAP one
* ...
* *OPT NFRM
* *MAP two
* ...
* will produce a form for MAP one but not for MAP two. By
moving the FRM/NFRM * lines you will be able to try the
various MAPs contained in this program
* one at a time or in batches. As delivered ALL the maps will
produce forms.
*
* The last MAP (named 'top') will produce a form
(topfrm.html) with links to * the other forms created.
*
**************************************************************
********
*OPTION HTMLC
*IFILE ARGS OR=CGI MAP=top
*DICTIONARY
 AREA     = 2/2         [ AREA
 ACNO     = 4/4         [ ACCOUNT NUMBER
 BALANCE  = 324:4       [ BALANCE

 AREAREQ  = 1000/2      [ AREA REQUIRED defined by CGI form
'read'
 TEXTAREA       /40     [ for textarea
 BRAZIL         /1      [ For checkbox
 CASHEW         /1
 PISTACHIO      /1
 AGECODE        /1      [ for radio
 AGERANGE       /1      [ for select
 PEANUT         /1      [ more checkbox items
 HAZELNUT       /1      [
 BETEL          /1      [
 ALMOND         /1      [
 WING           /1      [
* N.B.  there is no need for the variables 'lit' and 'link'
'submit' etc. as * they are not mapped to input fields.
```

```
*OPT FRM
*
* ==========================================================
iafrm.html
*MAP ia ACTION=topfrm.html,TITLE=TYPE=text example (single)
* ========================
*
*  i) a) single text input (no submit)
*
*
NAME=area                                 :AREAREQ
<BR><BR>Note\: no submit button is added for a single text
field  :lit
*
* ==========================================================
ibfrm.html
*MAP ib ACTION=topfrm.html,TITLE=TYPE=text
example(multiple)no breaks
* ========================
*
*  i) b) multiple text input (submit generated) no breaks
*
TYPE=text,NAME=area                          :AREAREQ
NAME=area,PROMPT=Enter area number           :AREAREQ
NAME=area,PROMPT=Area,VALUE=21               :AREAREQ
NAME=area,PROMPT=Area,SIZE=2,MAXLEN=2        :AREAREQ
<BR><BR>Note\: The above buttons are badly arranged because
we have not added any &lt;BR&gt;s to break the lines (see
link c)        :lit
*
* ==========================================================
icfrm.html
*MAP ic  ACTION=topfrm.html,TITLE=TYPE=text example
(multiple) with breaks
* =========================
*
*  i) c) multiple text input;  breaks added (2 methods)
TYPE=text,NAME=area                           :AREAREQ
<BR>                                          :useBRliteral
NAME=area,PROMPT=Enter area number            :AREAREQ
NAME=area,PROMPT=<BR>Area,VALUE=21            :AREAREQ
NAME=area,PROMPT=<BR>Area,SIZE=2,MAXLEN=2     :AREAREQ
<BR><BR>Note\: adding &lt;BR&gt;s to break the lines gives a
better layout   :lit
*
* ==========================================================
iifrm.html
*MAP ii  ACTION=topfrm.html,TITLE=TYPE=password
* =========================
*
```

```
* ii) password
*
TYPE=password,NAME=area,PROMPT=Area (secret),SIZE=2,MAXLEN=2
:AREAREQ
<BR><BR>Note\: Asterisks are displayed in place of typed
characters :lit
*
* ==========================================================
iiifrm.html
*MAP iii  ACTION=topfrm.html,TITLE=TYPE=textarea
* ==========================
*
* iii) textarea
*
This is textarea1 (default size)<BR>                  :lit
TYPE=textarea,NAME=textarea,PROMPT=textarea1
:TEXTAREA
This is textarea2 (size specified)<BR>                :lit
TYPE=textarea,NAME=textarea,PROMPT=textarea2,SIZE=5.5
:TEXTAREA
<BR>This is textarea3 (default contents)<BR>          :lit
TYPE=textarea,NAME=textarea,PROMPT=t3,VALUE=Type something
in here :TEXTAREA
*
*
============================================================
==== iv
* No MAP
* ======
*
*  iv) literals
*
* see other examples
** =========================================================
vfrm.html
*MAP v  ACTION=topfrm.html,TITLE=TYPE=image
* ========================
*
*   v) images
*
This form contains two images, one links to the gif file
<BR><BR> :lit
TYPE=image,SOURCE=logo.gif
:image
<A HREF\="logo.gif">                                    :lit
TYPE=image,SOURCE=logo.gif
:image
</A>                                                    :lit
*
* ========================================================
vifrm.html
```

```
*MAP vi  ACTION=topfrm.html,TITLE=TYPE=checkbox
* =========================
*
*  vi) checkboxes
*
This form contains three checkboxes.                  :lit
We've made Cashew 'ON' by default. <BR><BR>           :lit
TYPE=checkbox,NAME=BRAZIL   ,PRO=Brazil   ,VAL=Y
:BRAZIL
TYPE=checkbox,NAME=CASHEW   ,PRO=Cashew   ,VAL=Y,CHECK=Y
:CASHEW
TYPE=checkbox,NAME=PISTACHIO,PRO=Pistachio,VAL=Y
:PISTACHIO
*
* ========================================================
viifrm.html
*MAP vii ACTION=topfrm.html,TITLE=TYPE=radio
* =========================
*
* vii) radio
*
This form contains four radiobuttons.<BR><BR>           :lit
Enter your age range\:              <BR><BR>           :lit
TYPE=radio,NAME=age,PRO=under 19,VAL=A
:AGECODE
TYPE=radio,NAME=age,PRO=20-39   ,VAL=B,CHECKED=Y
:AGECODE
TYPE=radio,NAME=age,PRO=40-59   ,VAL=C
:AGECODE
TYPE=radio,NAME=age,PRO=over 60 ,VAL=D
:AGECODE
*
* =======================================================
viiifrm.html
*MAP viii ACTION=topfrm.html,TITLE=TYPE=select
* ==========================
*
* viii) select
*
This form contains three selections (pull-down
lists).<BR>:lit
<BR>Enter your age range\:              <BR><BR>        :lit
TYPE=select,NAME=age,VAL=A,PRO=under 19
:AGERANGE
TYPE=select,NAME=age,VAL=B,PRO=20's and 30's
:AGERANGE
TYPE=select,NAME=age,VAL=C,PRO=40's and 50's
:AGERANGE
TYPE=select,NAME=age,VAL=D,PRO=over 60
:AGERANGE
<BR><BR>Enter your age range (one has been preselected)
<BR><BR>:lit
```

```
TYPE=select,NAME=age2,VAL=A,PRO=under 19
:AGERANGE
TYPE=select,NAME=age2,VAL=B,PRO=20's and 30's
:AGERANGE
TYPE=select,NAME=age2,VAL=C,PRO=40's and 50's,SEL=Y
:AGERANGE
TYPE=select,NAME=age2,VAL=D,PRO=over 60
:AGERANGE
<BR><BR>Enter your age range (SIZE keyword used)
<BR><BR>:lit
TYPE=select,NAME=age3,VAL=A,PRO=under 19,SIZ=3
:AGERANGE
TYPE=select,NAME=age3,VAL=A,PRO=20's
:AGERANGE
TYPE=select,NAME=age3,VAL=A,PRO=30's
:AGERANGE
TYPE=select,NAME=age3,VAL=C,PRO=40's
:AGERANGE
TYPE=select,NAME=age3,VAL=B,PRO=50's
:AGERANGE
TYPE=select,NAME=age3,VAL=D,PRO=over 60
:AGERANGE
<BR><BR>Note\: select NAME change gives new selection.
<BR>:lit
*
* ==========================================================
ixfrm.html
*MAP ix ACTION=topfrm.html,TITLE=TYPE=submit
* ========================
*
* ix) submit
*
This form contains a submit (only one allowed)
<BR><BR>:lit
TYPE=submit,PRO=Press this>>>,VALUE=NOW!
:submit
<BR><BR>Note\: we have used PROMPT and VALUE keywords here
:lit
*
* ==========================================================
xfrm.html
*MAP x  ACTION=topfrm.html,TITLE=TYPE=reset
* ========================
*
*  x) reset
*
This form contains checkboxes and text boxes.<BR> We've made
Cashew 'ON' by default. <BR><BR> Change something and then hit
the reset button  <BR><BR> :lit
TYPE=checkbox,NAME=BRAZIL   ,PRO=Brazil   ,VAL=Y
:BRAZIL
TYPE=checkbox,NAME=CASHEW   ,PRO=Cashew   ,VAL=Y,CHECK=Y
:CASHEW
```

```
TYPE=checkbox,NAME=PISTACHIO,PRO=Pistachio,VAL=Y
:PISTACHIO
NAME=area,PROMPT=<BR><BR>Area,VALUE=No Area Input
:AREAREQ
TYPE=reset,PRO=<BR><BR>Press this to reset>>,VALUE=Undo
:reset
TYPE=submit                                             :submit
*
* ==========================================================
xiafrm.html
*MAP xia ACTION=topfrm.html,TITLE=TYPE=format (No
formatting)
* =========================
*
* xia) format
*
No formatting                                         <BR>:lit
TYPE=checkbox,NAME=BRAZIL   ,PRO=Brazil   ,VAL=Y
:BRAZIL
TYPE=checkbox,NAME=CASHEW   ,PRO=Cashew   ,VAL=Y
:CASHEW
TYPE=checkbox,NAME=PISTACHIO,PRO=Pistachio,VAL=Y
:PISTACHIO
TYPE=checkbox,NAME=PEANUT   ,PRO=Peanut   ,VAL=Y
:PEANUT
TYPE=checkbox,NAME=HAZELNUT ,PRO=Hazelnut ,VAL=Y
:HAZELNUT
TYPE=checkbox,NAME=BETEL    ,PRO=Betel    ,VAL=Y
:BETEL
TYPE=checkbox,NAME=ALMOND   ,PRO=Almond   ,VAL=Y
:ALMOND
TYPE=checkbox,NAME=WING     ,PRO=WING     ,VAL=Y        :WING
Enter your age range\:                                    :lit
TYPE=select,NAME=age,VAL=A,PRO=under 19
:AGERANGE
TYPE=select,NAME=age,VAL=B,PRO=20's and 30's
:AGERANGE
TYPE=select,NAME=age,VAL=C,PRO=40's and 50's
:AGERANGE
<BR><BR>Now we'll introduce <A
HREF\="xibfrm.html">formatting</A>:lit
*
* ==========================================================
xibfrm.html
*MAP xib ACTION=topfrm.html,TITLE=TYPE=format and format
changes
* ========================
*
* xib) format
*
* This is quite a complex form which demonstrates format
changes
*
```

```
*
We'll introduce a format using                         :lit
<PRE>TYPE\=format,VAL\=4</PRE>                         :lit
TYPE=format,VAL=4                                   :format
TYPE=checkbox,NAME=BRAZIL   ,PRO=Brazil   ,VAL=Y
:BRAZIL
TYPE=checkbox,NAME=CASHEW   ,PRO=Cashew   ,VAL=Y
:CASHEW
TYPE=checkbox,NAME=PISTACHIO,PRO=Pistachio,VAL=Y
:PISTACHIO
TYPE=checkbox,NAME=PEANUT   ,PRO=Peanut   ,VAL=Y
:PEANUT
TYPE=checkbox,NAME=HAZELNUT ,PRO=Hazelnut ,VAL=Y
:HAZELNUT
TYPE=checkbox,NAME=BETEL    ,PRO=Betel    ,VAL=Y
:BETEL
TYPE=checkbox,NAME=ALMOND   ,PRO=Almond   ,VAL=Y
:ALMOND
TYPE=checkbox,NAME=WING     ,PRO=WING     ,VAL=Y        :WING
Enter your age range\:                                 :lit
TYPE=select,NAME=age,VAL=A,PRO=under 19
:AGERANGE
TYPE=select,NAME=age,VAL=B,PRO=20's and 30's
:AGERANGE
TYPE=select,NAME=age,VAL=C,PRO=40's and 50's
:AGERANGE
TYPE=format,VAL=0                                   :format
<BR><BR>Now to try a different format. We switch off old
format\: :lit
<PRE>TYPE\=format,VAL\=0</PRE> and then introduce a new one\:
:lit
<PRE>TYPE\=format,VAL\=3</PRE>                         :lit
TYPE=format,VAL=3                                   :format
TYPE=checkbox,NAME=BRAZIL   ,PRO=Brazil   ,VAL=Y
:BRAZIL
TYPE=checkbox,NAME=CASHEW   ,PRO=Cashew   ,VAL=Y
:CASHEW
TYPE=checkbox,NAME=PISTACHIO,PRO=Pistachio,VAL=Y
:PISTACHIO
TYPE=checkbox,NAME=PEANUT   ,PRO=Peanut   ,VAL=Y
:PEANUT
TYPE=checkbox,NAME=HAZELNUT ,PRO=Hazelnut ,VAL=Y
:HAZELNUT
TYPE=checkbox,NAME=BETEL    ,PRO=Betel    ,VAL=Y
:BETEL
TYPE=checkbox,NAME=ALMOND   ,PRO=Almond   ,VAL=Y
:ALMOND
TYPE=checkbox,NAME=WING     ,PRO=WING     ,VAL=Y        :WING
Enter your age range\:                                 :lit
TYPE=select,NAME=age,VAL=A,PRO=under 19
:AGERANGE
TYPE=select,NAME=age,VAL=B,PRO=20's and 30's
:AGERANGE
```

```
TYPE=select,NAME=age,VAL=C,PRO=40's and 50's
:AGERANGE
TYPE=format,VAL=0                                            :format
<BR><BR>This isn't quite what we want. The prompt for the
:lit
selection shouldn't be in the last cell on the third
line.<BR>:lit
To improve the format we can do one of two things\:<BR>  :lit
<A HREF\="xicfrm.html#i">i) switch the format off (with
VAL\=0) and on again (with VAL\=2)  :lit
before the prompt</A><BR>                                   :lit
<A HREF\="xicfrm.html#ii">ii) have a blank cell before the
prompt</A> :lit
*
*
==========================================================xic
frm.html
*MAP xic ACTION=topfrm.html,TITLE=TYPE=format (and blank
cell)
* =========================
*
* xic) format
*
* This is quite a complex form which demonstrates format
changes
*
*
<A NAME\="xicfrm.html#i">i) switch the format off (with
VAL\=0) and on again (with VAL\=3**)  :lit
before the prompt</A><BR>                                   :lit
TYPE=format,VAL=3                                        :format
TYPE=checkbox,NAME=BRAZIL   ,PRO=Brazil   ,VAL=Y
:BRAZIL
TYPE=checkbox,NAME=CASHEW   ,PRO=Cashew   ,VAL=Y
:CASHEW
TYPE=checkbox,NAME=PISTACHIO,PRO=Pistachio,VAL=Y
:PISTACHIO
TYPE=checkbox,NAME=PEANUT   ,PRO=Peanut   ,VAL=Y
:PEANUT
TYPE=checkbox,NAME=HAZELNUT ,PRO=Hazelnut ,VAL=Y
:HAZELNUT
TYPE=checkbox,NAME=BETEL    ,PRO=Betel    ,VAL=Y
:BETEL
TYPE=checkbox,NAME=ALMOND   ,PRO=Almond   ,VAL=Y
:ALMOND
TYPE=checkbox,NAME=WING     ,PRO=WING     ,VAL=Y        :WING
TYPE=format,VAL=0                                        :format
TYPE=format,VAL=3                                        :format
Enter your age range\:                                      :lit
TYPE=select,NAME=age,VAL=A,PRO=under 19
:AGERANGE
TYPE=select,NAME=age,VAL=B,PRO=20's and 30's
:AGERANGE
```

```
TYPE=select,NAME=age,VAL=C,PRO=40's and 50's
:AGERANGE
TYPE=format,VAL=0                                         :format
<BR>** experiment with this value. The number is the number
:lit
of cells <I>across</I> the page so, if set to '2', the width
is  :lit
split into two equal cells, if '3', into thirds etc..<BR><BR>
:lit
<A NAME\="xicfrm.html#ii">ii) have a blank cell before the
prompt</A> :lit
TYPE=format,VAL=3                                         :format
TYPE=checkbox,NAME=BRAZIL   ,PRO=Brazil   ,VAL=Y
:BRAZIL
TYPE=checkbox,NAME=CASHEW   ,PRO=Cashew   ,VAL=Y
:CASHEW
TYPE=checkbox,NAME=PISTACHIO,PRO=Pistachio,VAL=Y
:PISTACHIO
TYPE=checkbox,NAME=PEANUT   ,PRO=Peanut   ,VAL=Y
:PEANUT
TYPE=checkbox,NAME=HAZELNUT ,PRO=Hazelnut ,VAL=Y
:HAZELNUT
TYPE=checkbox,NAME=BETEL    ,PRO=Betel    ,VAL=Y
:BETEL
TYPE=checkbox,NAME=ALMOND   ,PRO=Almond   ,VAL=Y
:ALMOND
TYPE=checkbox,NAME=WING     ,PRO=WING     ,VAL=Y        :WING
                                                    :blank
Enter your age range\:                                     :lit
TYPE=select,NAME=age,VAL=A,PRO=under 19
:AGERANGE
TYPE=select,NAME=age,VAL=B,PRO=20's and 30's
:AGERANGE
TYPE=select,NAME=age,VAL=C,PRO=40's and 50's
:AGERANGE
TYPE=format,VAL=0                                         :format
<BR><BR>Notes\:<BR>a) HTML ignores spacing so we have to use
&amp;nbsp; :lit
(no-break space) to force a blank cell
:lit
<BR><BR>b) to avoid having to type in
:lit
<PRE>TYPE\=format,VAL\=0                \:format</PRE> or
:lit
<PRE>&amp;nbsp;                               \:blank</PRE>
:lit
these lines could be held in *COPY files and copied into the
*MAP using (say) :lit
<PRE>*COPY formatoff</PRE> or <PRE>*COPY mapblank</PRE>
:lit
*OPT NFRM
*OPT FRM
*
```

```
* ==========================================================
topfrm.html
*MAP top TITLE=KEYWORDS for form generation
* ==========================================
*
*  Create a topfrm.html with links to rest of forms...
*  N.B. '\' to protect '=' in detail lines
*
This page has links to default forms generated by the
compiler.<BR>   :lit
If you have used *OPT NFRM above to suppress the generation
of<BR>    :lit
some forms then some of these links may be invalid.<BR>
:lit
<BR><B>TYPE\=text</B>:lit
<BR><A HREF\="iafrm.html">i) a) single  (no submit)</A>
:link
<BR><A HREF\="ibfrm.html">i) b) multiple (submit added) no
breaks</A> :link
<BR><A HREF\="icfrm.html">i) c) multiple (2 ways of adding
breaks)</A>:link
<BR><BR><B>TYPE\=password</B>
:lit
<BR><A HREF\="iifrm.html">ii) an example</A>
:link
<BR><BR><B>TYPE\=textarea</B>
:lit
<BR><A HREF\="iiifrm.html">iii) an example</A>
:link
<BR><BR><B>TYPE\=literals</B>
:lit
<BR> iv) See other examples
:lit
<BR><BR><B>TYPE\=images  </B>
:lit
<BR><A HREF\="vfrm.html">v) an example</A>
:link
<BR><BR><B>TYPE\=checkboxes</B>
:lit
<BR><A HREF\="vifrm.html">vi) an example</A>
:link
<BR><BR><B>TYPE\=radiobuttons</B>
:lit
<BR><A HREF\="viifrm.html">vii) an example</A>
:link
<BR><BR><B>TYPE\=select</B>
:lit
<BR><A HREF\="viiifrm.html">viii) an example</A>
:link
<BR><BR><B>TYPE\=submit</B>
:lit
<BR><A HREF\="ixfrm.html">ix) an example</A>
:link
```

```
<BR><BR><B>TYPE\=reset</B>
:lit
<BR><A HREF\="xfrm.html">x) an example</A>
:link
<BR><BR><B>TYPE\=format</B>
:lit
<BR><A HREF\="xiafrm.html">xi) a) No formatting</A><BR>
:link
<BR><A HREF\="xibfrm.html">xi) b) With formatting</A><BR>
:link
<BR><A HREF\="xicfrm.html">xi) c) Formatting techniques</
A><BR>       :link
TYPE=submit,VALUE=End of examples
:lit

*DETAB START
I DISPLAY 'Program for Compile Only'
*STOP
```

Appendix E
Informix C-ISAM

This appendix describes keywords, verbs and reserved words required to access Informix C-ISAM index sequential files.

E.1 Overview

C-ISAM supports:

- Split keys and multiple alternate indexes
- Index sequential file creation
- Record locking
- Key compression
- Transaction locking
- Logging to a named file

These facilities are made available through the use of keyword/value pairs in the **`*FILE`** or **`*?FILE`** directive.

Keyword/value pairs have the following features:

A keyword may not appear more than once. The list of keyword/values may be split over more than one line. A keyword/value pair may not be split. A keyword may be preceded by one or more spaces but the keyword/value pair may not contain spaces.

Keyword documentation appears in the following sections:

- *Organisation on page 2*
- *Record Length on page 2*
- *Keys on page 2*
- *Record Locking on page 4*
- *Transaction Locking on page 5*

E.1.1 Organisation

To define the type of file to be processed as C-ISAM, the OR parameter must be used

Organisation	
keyword: acceptable value:	**OR** C

E.1.2 Record Length

Record Length	
keyword: acceptable value: default:	**RL** a positive integer 80

The maximum length of records on the file is assumed to be that specified by the Record Length (**RL**) keyword. The value specifies the maximum number of bytes transferred from the file.

Minimum Record Length	
keyword: acceptable value: default:	**ML** a positive integer value associated with **RL**

The minimum record length for variable length record files.

E.1.3 Keys

Primary Key	
keyword: acceptable values: default:	**PK** start type length OR (start type length {,start type length}) where: start and length are positive integers, type is a FILETAB field type (**:;/**) None

Split keys are supported here. Each part of the key is specified using it's start, type and length. Multiple specifications must be separated by a comma with the entire list being enclosed in parentheses.

Note: The value of start is taken as the offset into the record, and therefore starts at 0.

Note: {,xxx} indicates that, xxx can occur 0 or more times. The { and } do not appear in code.

Examples

PK=0/10 - contiguous key
PK=20:4 - contiguous key
PK=(0/10, 20:4) - split primary key

Alternate Key n	
keyword: acceptable values: default:	**AKn** start type length OR (start type length {,start type length}) where: start and length are positive integers type is a FILETAB field type (:;/) None

These keyword/value pairs specify alternate keys.

Primary Key Compression	
keyword: acceptable values: default:	**PC** D, L, T, M None

Alternate Key n Compression	
keyword: acceptable values: default:	**ACn** D, L, T, M None

D Duplicates are compressed
L Leading characters are compressed
T Trailing characters are compressed
M Maximum possible compression is used

Primary Key Duplicates	
keyword: acceptable value: default:	**PD** Y No

Alternate Key n Duplicates	
keyword: acceptable value: default:	**ADn** Y No

E.1.4 Record Locking

Record Locking Method	
keyword: acceptable values: default depends on file type:	**LM** A, M, E, W ***FILE**, ***IFILE** : None ***OFILE** : E other : A

The following locking methods are available:

A Automatic: **READ**ing a record locks it. The lock is released when another operation is carried out

M Manual: **READ**ing a record locks it (***IOFILE** or ***EFILE** only). The locks are only removed by calling **RELEASE**, **COMMIT** or **ROLLBACK**

E Exclusive: The file is locked when it is opened. If another application is accessing the file at the time then a run-time error will occur

W Wait: When attempting a read, the application waits for locked records to be unlocked. Wait mode can be used in conjunction with the other methods (for example, **LM=A**, **LM=W**). If used alone it affects default behaviour

The default behaviour depends on the file type.

File Type	Default Behaviour
***FILE** ***IFILE**	No locking. Records locked by other applications can still be read
***OFILE**	Exclusive
all others	Automatic no wait. A **READ** will skip a record (return RECLEN = 0) if it is already locked. This will return FALSE if conditional **READ**

E.1.5 Transaction Locking

Use Logfile	
keyword: acceptable value: default:	**UL** logfile None

The use of this keyword indicates that transaction locking is to be used. Transactions will be written to a log file. The log file will only be created for a ***OFILE** or ***IOFILE** which does not exist (that is, the C-ISAM file is also created by FILETAB).

Create Logfile	
keyword: acceptable value: default:	**LF** logfile None

This keyword is the same as **UL** except that log files will always be created if they do not already exist.

E.2 Verbs and C-ISAM Files

E.2.1 COMMIT File

Use this verb to:

- Write the changes made (by **`WRITE`** verbs) to file and any other files involved in the transactions
- Unlock all locked records in all files involved in the transactions
- Record the transactions in the associated logfile
- Start a new transaction

It should only be used when transaction logging has been specified.

E.2.2 ROLLBACK File

Use this verb to:

- Undo any changes made (by **`WRITE`** verbs) to file and any other files involved in the transactions
- Unlock all locked records in all files involved in the transactions
- Start a new transaction

It should only be used when transaction logging has been specified.

E.2.3 RELEASE File

Use this verb to unlock all locked records in file.

Note: If transaction logging has been specified any changed records will not be unlocked by this verb (until execution of a **`COMMIT`** *or* **`ROLLBACK`***).*

E.2.4 INDEX File n

Use this verb to select the nth index (0 = primary) for subsequent accesses to file.

E.3 Reserved Words and C-ISAM Files

E.3.1 ZREPLY

This reserved word is used to report the status of C-ISAM file accesses. It is set to the C-ISAM error number if any C-ISAM error has occurred. Otherwise it is set to zero.

Warnings are indicated by negative returns, errors by positive returns.

A selection of frequently encountered returns are given in the following tables.

ERROR Value	ERROR Description
100	Inserting a duplicate key into an index which cannot contain duplicates
107	Accessing a record or file locked by another application

WARNING Value	WARNING Description
-100	**WRITE** has inserted a duplicate key into an index
-101	**READ** with more records with the same key in the current index (including primary)

E.4 Field Types Supported With C-ISAM

In addition to the standard field types supported by FILETAB the C-ISAM interface supports two more field types:

- **External Decimal**
  These are INFORMIX decimal / fixed point / money fields as held on C-ISAM files.
- **Internal Decimal**
  The internal version of these fields, on which operations can be carried out.

Note: The C-ISAM library `libisam.a` *must be licensed for and installed on your machine for these types to be supported.*

E.5 Internal Decimal Fields

These are fields which hold decimal (floating point) numbers and on which arithmetic and comparison operations can be performed. These fields are variously described as: decimal, fixed precision, money, and numeric.

E.5.1 Field Definition

Internal Decimal fields are defined using the basic format

start-position:24<INT_DEC>

Any of the alternative field definition formats can be used apart from those containing (indirect-length).

E.5.2 Operations and Internal Decimal Fields

The operations which can be applied to Internal Decimal fields are:

- Arithmetic Operations
- Comparison Operations
- Move Operation (**MV**)

In each case the second operand is converted to the type of the first operand (see below for details).

- Zeroise Operation

Operations which cannot be applied to Internal Decimal fields are: One-of-a-set, Spacefill, Fill, Logical operations and Multiple Move.

E.5.3 Conversion To and From Internal Decimal Fields

- When converting a Internal Decimal field to a binary, or unsigned field the decimal value will be rounded
- When converting a binary, or unsigned field to a Internal Decimal field the result will be a whole number
- Date fields are treated as binary fields containing the number of days since 31 December 1899
- When converting to and from Floating Point fields some precision may be lost, as Floating Point fields contain less significant digits but can represent larger (or smaller) values
- When moving a character field, or literal, to a Internal Decimal field it will be converted if it contains a numeric constant consisting of a number optionally preceded by a + or − and optionally followed by a `.` and another number, for example, `-345.67`. Leading spaces are allowed and trailing spaces ignored

- When moving a Internal Decimal field to a character field it will be converted into a character format. By default only the whole number part will be output and the sign will be ignored. To specify that the sign is output then set ***OPT NIS**. To specify that a number of decimal places are output use ***OPTION DP** as described with Floating Point fields. If too many characters are produced a run-time error will occur (or if ***OPT NBME** is set the field will be filled with *****s) , if too few characters are produced they will be padded to the left with spaces

E.5.4 Numeric Constants

Numeric Constants when used with Internal Decimal fields can consist of a number optionally preceded by a + or − and optionally followed by a . and another number. For example:

```
-345.67
```

E.5.5 Associating Internal Decimal Fields with FSCs in the *INLIST (or a *LIST)

An Internal Decimal field can be associated with a Transfer, Control or Totalling FSC. Only one field should be associated with each FSC.

The default Totalling FSCs are binary fields. To make all totalling FSCs Internal Decimal fields then use ***OPT FSCT=10**. To change specific Totalling FSCs to use Internal Decimal field then use the ***ALTER FSCT10** directive, for example

```
*ALTER FSCT10  [Change 1 2 and 3 to total
123            [using Internal Decimal Fields
```

E.5.6 Printing of FSCs associated with Internal Decimal Fields

In ***OUT**s and ***HEAD**s FSCs mapped onto Internal Decimal fields are treated in the same way as FSCs mapped onto Floating Point fields.

E.6 External Decimal Fields

These are INFORMIX decimal fields as they are held on a data file, which is as a packed form of Internal Decimal.

E.6.1 Field Definition

External Decimal fields are defined using the basic format:

```
start-position/length<EXT_DEC>
```

Any of the alternative field definition formats can be used. The length is the size of the field on the file (or its width). This can be calculated as the number of significant digits (specified) divided by 2, and rounded up,

plus 1. (In Sea-Change the standard length / width is 11 for fixed point and money).

E.6.2 Treatment of External Decimal Fields

External Decimal fields are handled as follows:

- When an External Decimal field is moved to an Internal Decimal field its contents are converted to the internal format
- When an Internal Decimal field is moved to an External Decimal field its contents are converted to the external format. Note that the length of the External Decimal field should be enough to hold the number
- For all other operations on a External Decimal field it is automatically converted to and / or from an Internal Decimal field
- External Decimal fields can be mapped to Control, Transfer and Totalling FSCs however:
  - To retain the fractional part of External Decimal fields the Totalling FSCs used should those specified to use Internal Decimal (or Floating Point) fields
  - For efficiency improvements External Decimal fields should not be mapped directly to Control or Transfer FSCs

Note: Additionally for efficiency improvements Arithmetic and Comparison operations should not be used directly on External Decimal fields. Instead the External Decimal fields should be moved to Internal Decimal fields which can then be further manipulated (or passed to the fixed logic).

Appendix F
ODBC Guide

F.1 What Your ODBC Guide Contains

The *ODBC Guide* provides comprehensive details of how to access data held in an ODBC compliant database and is divided into the following sections:

F.1 What Your ODBC Guide Contains	Describes what is available in this document and how it relates to the FILETAB software
F.2 Keywords and ODBC Files	Explains the keywords available with ODBC and how to use them
F.3 Directives and ODBC Files	Explains the directives available with ODBC and how to use them
F.4 Verbs and ODBC Files	Explains the verbs available with ODBC and how to use them
F.5 Reserved Words and ODBC Files	Explains the reserved words available with ODBC and how to use them
F.6 Field Types Supported with ODBC	Explains the field types supported with ODBC and how to use them
F.7 Examples	Demonstrates how to use ODBC

F.2 Keywords and ODBC Files

ODBC databases may be accessed by `*FILE` or `*?FILE` directives which have an `ODBC` organisation.

To access such files the SQL which will be used to extract information must be specified in a `*SQL` directive. This must be associated with the file by use of a keyword.

Additionally indicator variables can be specified in an `*INDICATORS` directive and associated with the file via a keyword.

A `*DETAB` for handling errors may also be specified via a keyword.

F.2.1 Organisation

Keyword	OR
Value	ODBC

F.2.2 SQL Directive Name

Keyword	SQL
Value	sql_directive_name

This will associate the *SQL directive sql_directive_name with the *?FILE name. The SQL in the named directive will be used to determine the view of the database on which file handling verbs (and fixed logic) operate. The named SQL directive should only contain a single SQL statement.

F.2.3 Record Length

Keyword	RL
Value	a positive integer
Default	NONE

This specifies the size of the data area returned by the SQL directive above. The size of the area must match the record length **exactly**. See below for details of the FILETAB field types which should be used for ODBC column types. At run-time if it is found that this value is incorrect a suitable *DICTIONARY is created in the file ODBCDATA.DIC and the run is aborted. However due to differing naming conventions between FILETAB and the ODBC database this file may require editing.

F.2.4 INDICATORS Directive Name

Keyword	**IND**
Value	ind_directive_name

This associates the *INDICATORS directive ind_directive_name with the *?FILE name. The *INDICATORS directive will be used to determine the indicator variables for columns read from the database.

F.2.5 Error Handling *DETAB

Keyword	**ONERROR**
Value	detab_name

This specifies a *DETAB which will be called whenever an error or warning condition occurs whilst processing the file.

F.2.6 Database

Keyword	**NAME**
Value	database_name

This will specify the database name. The database name can also be specified by the usual file naming methods. The database name maps onto a valid ODBC DSN.

F.2.7 User

Keyword	**USER**
Value	`username/password`

If security has been enabled on the ODBC database this will specify the user name/password which is used to connect to the database. If this keyword is omitted and security has been enabled then the system will produce a username/password dialogue box.

F.2.8 Row Cache

Keyword	**RC**
Value	`a positive integer`
Default	`100`

This specifies the number of rows to be cached when reading from the ODBC database

F.3 Directives and ODBC Files

F.3.1 *SQL Directive

Syntax:

```
*SQL name
SQL statements
```

This allows SQL statements to be specified, then used to retrieve columns from a database via a `READ` verb or by the fixed logic. It can also be used to specify SQL statements which can be executed using an `EXECSQL` verb.

FILETAB Host Fields can be included in the SQL statements. A Host Field will consist of a : followed by a FILETAB field name.

Host Fields allow you to specify FILETAB fields whose contents will be substituted into the SQL (when it is executed) as SQL literal values. This allows you to specify selection criteria, and so on, at run time. FILETAB `/` fields will be treated as string literals, `:`, `;`, `$` and other numeric fields as numeric literals, date fields as (`:4<DATE>`).

Optionally a second `:` and FILETAB field name can be placed after the first; the second field will act as an indicator variable.

Indicator variables are used to indicate if a field contains a `NULL` value. To indicate a `NULL` value the appropriate indicator variable should be set to less than zero.

If a `/` host field is specified in the first position of the SQL statement then the surrounding quotes of the string literal are omitted thus allowing an entire SQL statement to be built up at run-time.

F.3.2 *INDICATORS Directive

Syntax:

```
*INDICATORS name
{Indicator_Definition}
```

where:

`Indicator_Definition`

is:

`Column_Name : FILETAB_Field_Name`

This directive is used to specify FILETAB fields, which will be used as indicator variables for columns retrieved using the `READ` verb or via the fixed logic. The `Column_Name` must conform to ODBC naming conventions. The `FILETAB_Field_Name` should be a `:4` field. The `*INDICATORS` directive is associated with a file using the `IND` keyword.

When data is retrieved, if a named `Column_Name` contains a `NULL` value its associated `FILETAB_Field_Name` will be set to less than zero.

F.4 Verbs and ODBC Files

F.4.1 CONNECT file_name

This verb, when executed, will attempt to connect to the database given via the `file_name`.

F.4.2 DISCONNECT file_name

This verb, when executed, will attempt to disconnect from the database given via the `file_name`.

F.4.3 CLOSE file_name

This verb will close the file specified. The database will **not** be disconnected (this will happen at the end of the run unless a `DISCONNECT` is used).

F.4.4 READ file_name Field

This verb (which will also be called by the fixed logic) will retrieve the rows specified by the `*SQL` associated with the `file_name`, one at a time, into the area specified by the field

Note: The file will be automatically opened and the database connected, if required.

F.4.5 EXECSQL file_name sql_directive_name

This verb will execute the SQL in the `*SQL` directive `sql_directive_name` on the database specified by the file

file_name. **EXECSQL will automatically load the database if required**. Any preparable SQL may be specified in the *SQL directive

F.5 Reserved Words and ODBC Files

The following reserved words are available for obtaining information about the latest executed SQL:

Reserved Word	Explanation
ODBCCODE:4	Is set to 0 if an SQL statement executes OK otherwise set to the ODBC error number
ODBCERRM/255	ODBC error message text

These reserved words are not available until after the first access of the database.

F.6 Field Types Supported with ODBC

The FILETAB field types which correspond to ODBC SQL column types are:

ODBC SQL Type	FILETAB Type	Type
CHAR(n), VARCHAR(n), DECIMAL(n), NUMERIC(n)	/n (max 255)	Character
If *OPT NUMDBL and *OPT INITOPT appear before *DICTIONARY DECIMAL (n) NUMERIC (n)	$8	Floating point
DATE, TIME, TIMESTAMP	/19	Character
BIT	;1	Unsigned integer
TINYINT	:1	Signed integer
SMALLINT	:2	Signed integer
INTEGER	:4	Signed integer
BIGINT	:8	Signed integer
REAL	$4	Floating point
DOUBLE, FLOAT	$8	Floating point
MEMO(n)	/n (max 1023)	Character

*Note: If you specify an incorrect record length for a file then a *DIC will be produced, into a file called "ODBCDATA.DIC", giving the fields type and length. To change the file name set the shell variable "FTC_ODBCDATA_DIC" to the required name ("" for the standard error output).*

F.7 Examples

Two examples demonstrate how FILETAB programs can be used to access an ODBC compliant database. They are based on a database

consisting of tables holding sample data for Sales Representatives, Customers, Orders, Order Lines and Stock Items.

- *Example 1 Sales Analysis on page 6*
  Produces a simple sales analysis based on a specified date range input by the user on the command line when running the program.
- *Example 2 Despatch Note on page 8*
  Produces despatch notes for allocated stock items and then updates the database accordingly.

F.7.1 Example 1 Sales Analysis

```
*PROGRAM EXAMPLE1                                [START OF PROGRAM
*
*OPTION PL=80,OC                               [PRINT LENGTH, OMIT
COMMENTS
*
*FILE SALES                                  [MAIN FILE LOCAL NAME
          OR=ODBC                                [DATABASE
ORGANISATION
          NAME='FT Examples'                     [ODBC DATASOURCE
NAME (DSN)
          RL=95                                  [RECORD LENGTH
         SQL=SALESQUERY                        [*SQL STATEMENT NAME
*
*IFILE ARGS                                      [INPUT FILE LOCAL
NAME
          OR=CLI                                 [COMMAND LINE
ARGUMENTS
          MAP=ARGSMAP                           [ARGUMENT *MAP NAME
*
*DICTIONARY                                      [DATA DICTIONARY
  REGION      = /10                              [SALES REGION
  NAME       = /30                           [SALES REPRESENTATIVE
  ITEM        = /6                               [ITEM CODE
  DESCRIPTION = /30                              [ITEM DESCRIPTION
  SALES       = /19                              [SALES VALUE
*
  DATEARGS = /16                               [CLI ARGUMENTS INPUT
  FROMDATE = DATEARGS/8                          [MAP TO DATE FROM
  TODATE   = /8                                  [AND DATE TO
  DATEFROM = :4<DATE>                          [CONVERTED DATE FROM
  DATETO   = :4<DATE>                            [AND DATE TO
  VALUE    = :8                                  [CONVERTED SALES
VALUE
  P1       = :4                                  [POINTER
*
*INLIST                                          [INLIST AREA
  N REGION                                       [GROUP BY SALES
REGION
```

```
  M NAME                                [AND THEN SALES
REPRESENTATIVE
  A FROMDATE TODATE                     [DATE SELECTION
RANGE
  B ITEM                                [ITEM DETAILS
  C DESCRIPTION
  1 VALUE                             [SALES VALUE (TO BE
TOTALED)
*
*HEAD L,N CH1,1                         [HEADING FOR EACH
REGION (& PAGE)
Run on DD/MM/YYYY
Page PPPP

         'NNNNN'Regional Sales Analysis
for'AAAAAAAA'to'AAAAAAAA'

 Sales Representative     Item/Description
Sales
*
*HEAD M 1,0 [HEADING FOR EACH SALES REP.
'MMMMMMMMMMMMMMMMMMMMMMMMMMM'
*
*OUT L 0,1                            [OUTPUT FOR EACH ITEM
                           BBBBBB/
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC 11111.11
*
*OUT M 0,1                              [TOTAL OUTPUT FOR
EACH SALES REP.
                       'All Items'
11111.11
*
*OUT N 1,1                              [TOTAL OUTPUT FOR
EACH REGION
 NNNNN'Regional Sales Total'
11111.11
*
*OUT F 1,0                             [GRAND TOTAL OUTPUT
'Regional Sales Grand Total'
11111.11
*
*MAP ARGSMAP                            [CLI ARGUMENTS MAP
  :FROMDATE
  :TODATE
*
*SQL SALESQUERY
SELECT Region, Sales.Name, Item, Description, Sum(Value)
AS Sales
FROM (((Items INNER JOIN Lines ON Items.ID = Lines.Item)
             INNER JOIN Orders ON Lines.Order = Orders.ID)
              INNER JOIN Customers ON Orders.Customer =
Customers.ID)
```

```
            INNER JOIN Sales ON Customers.Sales = Sales.ID
WHERE Ordered BETWEEN #:DATEFROM# AND #:DATETO#
GROUP BY Region, Sales.Name, Item, Description
*
*DETAB START                                [PROCESS AT START
OF RUN
I READ ARGS DATEARGS/0                     [READ CLI ARGUMENTS
  DATEFROM MV FROMDATE                      [CONVERT DATE FROM
  DATETO MV TODATE                          [AND DATE TO
*
*DETAB RECORD                               [PROCESS FOR EACH
RECORD
I VALUE MV SALES/14                       [CONVERT SALES VALUE
POUNDS
  VALUE * 100                               [CONVERT TO PENCE
  P1 MV A'SALES                            [POINT TO START OF
SALES VALUE
  P1 ST '.'                               [SKIP TO THE DECIMAL
POINT
  P1 + 1                                    [AND THEN OVER IT
  VALUE + (P1)/2                            [ADD IN THE SALES
VALUE PENCE
*
*STOP                                       [END OF PROGRAM
```

F.7.2 Example 2 Despatch Note

```
*PROGRAM EXAMPLE2[START OF PROGRAM
*
*OPTION PL=80,OC                  [PRINT LINE LENGTH, OMIT COMMENTS
*
*FILE DESPATCH                    [MAIN FILE LOCAL NAME
  OR=ODBC                      [DATABASE ORGANISTATION
  NAME='FT Examples'           [ODBC DATASOURCE NAME (DSN)
  RL=265                       [RECORD LENGTH
  SQL=SENDITEMS                [*SQL STATEMENT NAME
*
*IFILE UPDATES                    [UPDATE FILE LOCAL NAME
  OR=ODBC                      [DATABSE ORGANISATION
  NAME='FT Examples'           [ODBC DATASOURCE NAME (DSN)
  RL=1                         [DUMMY RECORD LENGTH
  SQL=UPDATEITEMS              [*SQL STATEMENT NAME
*
*DICTIONARY                            [DATA DICTIONARY
  CUSTID      = :4                     [CUSTOMER ID
  CUSTNAME    = /30                    [CUSTOMER NAME
  ADDRESS_1   = /30                    [ADDRESS LINE 1
  ADDRESS_2   = /30                    [ADDDRES LINE 2
  ADDRESS_3   = /30                    [ADDRESS LINE 3
  ADDRESS_4   = /30                    [ADDRESS LINE 4
```

```
  POST_CODE   = /10                     [POST CODE
  SALENAME    = /30                     [SALES REPRESENTATIVES NAME
  ORDERID     = :4                      [ORDER ID
  ORDERED     = /19                     [ORDER DATE
  ITEM        = /6                      [ITEM CODE
  DESCRIPTION = /30                     [ITEM DESCRIPTION
  QUANTITY    = :4                      [QUANTITY ORDERED
  ALLOCATED   = :4                      [QUANTITY ALLOCATED FROM STOCK
  DESPATCHED  = :4                      [QUANTITY ALREADY DESPATCHED
*
  ORDERNO     = /4                      [ZERO FILLED ORDER ID
  ORDERDATE   = /10                     [TIDIED ORDER DATE
  TOFOLLOW    = :4                      [CALCULARED QUANTITY TO FOLLOW
*
*INLIST                                 [INLIST AREA
  N CUSTID                              [GROUP BY CUSTOMER
  M ORDERNO                             [AND THEN ORDER
  A CUSTNAME                            [CUSTOMER DETAILS
  B ADDRESS_1
  C ADDRESS_2
  D ADDRESS_3
  E ADDRESS_4
  F POST_CODE
  G SALENAME                            [SALES REPRESENTATIVE DETAILS
  H ORDERDATE                           [ORDER DETAILS
  I ITEM                                [ITEM DETAILS
  J DESCRIPTION
  1 QUANTITY                          [QUANTITY DETAILS (TO BE TOTALED)
  2 DESPATCHED
  3 ALLOCATED
  4 TOFOLLOW
*
*HEAD N CH1,1                           [HEADING FOR EACH CUSTOMER
Despatch to:                       Your Representative:
'
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         GGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
  EEEEEEEEEEEEEEEEEEEEEEEEEEEEEE       'Despatch Date:'
  FFFFFFFFFF                             'DD/MM/YYYY
     ----- Qty -----
Order No  Order Date  Item/Description Ord Snt Enc Flw
*
*HEAD M 1,0                             [HEADING FOR EACH ORDER
' MMMM    HHHHHHHHHH
*
*OUT L 0,1                              [OUTPUT FOR EACH ITEM
            IIIII/JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ  111 222 333 444
```

```
*
*OUT N 1,0                              [TOTAL OUTPUT FOR EACH CUSTOMER
 'All Orders'                                           111 222 333 444
*
*SQL SENDITEMS
SELECT Customers.ID, Customers.Name, [Address 1], [Address 2],
[Address 3], [Address 4], [Post Code], Sales.Name, Orders.ID,
Ordered, Item, Description, Quantity, Allocated, Lines.Despatched
FROM (((Lines INNER JOIN Items ON Lines.Item = Items.ID)
      INNER JOIN Orders ON Lines.Order = Orders.ID)
      INNER JOIN Customers ON Orders.Customer = Customers.ID)
      INNER JOIN Sales ON Customers.Sales = Sales.ID
WHERE Allocated > 0
*
*SQL UPDATEITEMS
UPDATE Lines SET Allocated = 0, Despatched = [Despatched]+[Allocated]
WHERE Allocated > 0
*
*DETAB START                            [PROCESS AT START OF RUN
I ORDERDATE MV '  /  /    '             [INITIALISE ORDER DATE
*
*OPTION NZS                             [DO NOT SUPPRESS ZEROS
*
*DETAB RECORD                           [PROCESS FOR EACH RECORD
I ORDERNO MV ORDERID                    [ZERO FILL ORDER ID
  ORDERDATE/2 MV ORDERED+8/2            [TIDY ORDER DATE
  ORDERDATE+3/2 MV ORDERED+5/2
  ORDERDATE+6/4 MV ORDERED/4
  TOFOLLOW MV QUANTITY                  [CALCULATE QUANTITY TO FOLLOW
  TOFOLLOW - DESPATCHED
  TOFOLLOW - ALLOCATED
*
*SORT *N,*M,*I                         [SORT BY CUSTOMER, ORDER AND ITEM
*
*DETAB BREAK F                          [PROCESS AT END OF RUN
I EXECSQL UPDATES UPDATEITEMS           [UPDATE ITEMS
*
*STOP                                   [END OF PROGRAM
```

Appendix G
Informix Guide

G.1 About your Informix Guide

The *Informix Guide* provides comprehensive details of how to access data held in an Informix compliant database and is divided into the following sections:

G.1 About your Informix Guide	Describes what is available in this document and how it relates to the FILETAB software
G.2 Keywords and Informix Files	Explains the keywords available with Informix and how to use them
G.3 Directives and Informix Files	Explains the directives available with Informix and how to use them
G.4 Verbs and Informix Files	Explains the verbs available with Informix and how to use them
G.5 Reserved Words and Informix Files	Explains the reserved words available with Informix and how to use them
G.6 Field Types Supported with Informix	Explains the field types supported with Informix and how to use them
G.7 Examples	Demonstrates how to use Informix

G.2 Keywords and Informix Files

Informix databases may be accessed by `*FILE` or `*?FILE` directives which have an `Informix` organisation. To access such files the SQL which will be used to extract information must be specified in a `*SQL` directive. This must be associated with the file by use of a keyword.Additionally indicator variables can be specified in an `*INDICATORS` directive and associated with the file via a keyword. A `*DETAB` for handling errors may also be specified via a keyword.

G.2.1 Organisation

Keyword	**OR**
Value	Informix

In order to enable the Informix organisation, the following UNIX or Linux environment variable should be set.

Depending on the shell, enter the following.

In case of csh	setenv FTC_FLAGS "-DInformix"
In case of ksh	FTC_FLAGS="-DInformix"; export FTC_FLAGS

G.2.2 SQL Directive Name

Keyword	**SQL**
Value	sql_directive_name

This will associate the *SQL directive sql_directive_name with the *?FILE name. The SQL in the named directive will be used to determine the view of the database on which file handling verbs (and fixed logic) operate. The named SQL directive should only contain a single SELECT statement.

G.2.3 Record Length

Keyword	**RL**
Value	a positive integer
Default	NONE

This specifies the size of the data area returned by the SQL directive above. The size of the area must match the record length **exactly**. When calculating the data area size the following points should be considered:

- Columns defined as DATE, INTEGER, INT, SMALLFLOAT, REAL and SERIAL occupy four bytes and will always start on the next 4 byte boundary
- Columns defined as FLOAT or DOUBLE PRECISION occupy eight bytes and will always start on the next 4 byte boundary
- Columns defined as DECIMAL, DEC, NUMERIC or MONEY occupy twenty four bytes and will always start on the next 4 byte boundary
- Columns defined as SMALLINT will occupy two bytes and will always start on the next 2 byte boundary
- Columns defined as VARCHAR will occupy one more byte than their maximum length

- Columns defined as CHARACTER will occupy their length

See below for details of the FILETAB field types which should be used for Informix column types, *OPT ALI can be used to align fields in a *DICTIONARY automatically.

G.2.4 INDICATORS Directive Name

Keyword	**IND**
Value	ind_directive_name

This associates the *INDICATORS directive ind_directive_name with the *?FILE name. The *INDICATORS directive will be used to determine the indicator variables for columns read from the database.

G.2.5 Error Handling *DETAB

Keyword	**ONERROR**
Value	detab_name

This specifies a *DETAB which will be called whenever an error or warning condition occurs whilst processing the file.

G.2.6 Database

Keyword	**NAME**
Value	database_name

This will specify the database name. The database name can also be specified by the usual file naming methods.

G.3 Directives and Informix Files

G.3.1 *SQL Directive

Syntax:

```
*SQL name
SQL statements
```

This allows SQL statements to be specified, then used to retrieve columns from a database via a READ verb or by the fixed logic. It can also be used to specify SQL statements which can be executed using an EXECSQL verb.

FILETAB Host Fields can be included in the SQL statements. A Host Field will consist of a : followed by a FILETAB field name.

Host Fields allow you to specify FILETAB fields whose contents will be substituted into the SQL (when it is executed) as SQL literal values. This allows you to specify selection criteria, and so on, at run time. FILETAB `/` fields will be treated as string literals, `:`, `;`, `$` and other numeric fields as numeric literals, date fields as (`:4<DATE>`).

Optionally a second `:` and FILETAB field name can be placed after the first; the second field will act as an indicator variable.

Indicator variables are used to indicate if a field contains a `NULL` value. To indicate a `NULL` value the appropriate indicator variable should be set to less than zero.

If a `/` host field is specified in the first position of the SQL statement then the surrounding quotes of the string literal are omitted thus allowing an entire SQL statement to be built up at run-time.

G.3.2 *INDICATORS Directive

Syntax:

```
*INDICATORS name
{Indicator_Definition}
```

where:

```
Indicator_Definition
```

is:

```
Column_Name : FILETAB_Field_Name
```

This directive is used to specify FILETAB fields, which will be used as indicator variables for columns retrieved using the `READ` verb or via the fixed logic. The `Column_Name` must conform to Informix naming conventions. The FILETAB fields should be `:2` fields. The `*INDICATORS` directive is associated with a file using the IND keyword.

When data is retrieved, if a named column contains a `NULL` value its indicator variable will be set to less than zero.

G.4 Verbs and Informix Files

G.4.1 CONNECT file_name

This verb, when executed, will attempt to connect to the database given via the `file_name`.

G.4.2 DISCONNECT file_name

This verb, when executed, will attempt to disconnect from the database given via the `file_name`.

G.4.3 CLOSE file_name

This verb will close the file specified. The database will **not** be disconnected (this will happen at the end of the run unless a DISCONNECT is used).

G.4.4 READ file_name Field

This verb (which will also be called by the fixed logic) will retrieve the rows specified by the *SQL associated with the file_name, one at a time, into the area specified by the field

Note: The file will be automatically opened and the database connected, if required.

G.4.5 EXECSQL file_name sql_directive_name

This verb will execute the SQL in the *SQL directive sql_directive_name on the database specified by the file file_name. **EXECSQL will automatically load the database if required**. Any preparable SQL may be specified in the *SQL directive

G.5 Reserved Words and Informix Files

The following reserved words are available for obtaining information about the latest executed SQL:

Reserved Word	Explanation
SQLCODE:4	Is set to 0 if an SQL statement executes OK, 100 if a SELECT retrieves nothing, or less than zero on an error
SQLERRM/70	Error message text
SQLERRD0:4	Estimated number of rows returned
SQLERRD1:4	Serial value after insert or ISAM error code
SQLERRD2:4	Number of rows processed
SQLERRD3:4	Estimated cost
SQLERRD4:4	Offset of the error into the SQL statement
SQLERRD5:4	rowid after insert
SQLWARN0/1	W if any of sqlwarn[1-5] reserved words contain W
SQLWARN1/1	W if any truncation occurred or database has transactions
SQLWARN2/1	W if a null value returned or ANSI database

Reserved Word	Explanation
SQLWARN3/1	W if the number of columns in the select-list is not equal to the number of items in the INTO list of the SELECT
SQLWARN4/1	W if no WHERE clause on prepared UPDATE, DELETE or incompatible float format
SQLWARN5/1	W if non-ANSI statement

These reserved words are only available after the database has first been accessed.

G.6 Field Types Supported with Informix

The FILETAB field types which correspond to Informix SQL column types are::

Informix SQL Type	FILETAB Field	Type
CHAR(n), CHARACTER(n)	/n	Character
VARCHAR(m,x)	/m	Character
DATE	:4<DATE>	Date
SMALLINT	:2	Integer
INTEGER, INT, SERIAL	:4	Integer
SMALLFLOAT, REAL	$4	Floating point
FLOAT, DOUBLE PRECISION	$8	Floating point
DECIMAL, DEC, NUMERIC, MONEY	:24<INT_DEC>	Internal decimal
BYTE, TEXT, DATETIME, INTERVAL		Not yet supported

Note: The Informix library `libsql.a` *must be licensed for and installed on your machine for the Internal Decimal type to be supported.*

G.6.1 How FILETAB Prints Informix NULL Fields

Informix SQL Type	FILETAB Field	Print Values
CHAR(n), CHARACTER (n)	/n	Rest of the line is blanked out
VARCHAR(m,x)	/m	A blank field
DATE	:4<DATE>	83/01/1899 (On UNIX) -214748328 (On Linux)
SMALLINT	:2	-32,768
INTEGER, INT, SERIAL	:4	-2,147,483,648
SMALLFLOAT, REAL	$4	Field filled with *'s
FLOAT, DOUBLE PRECISION	$8	Field filled with *'s
DECIMAL, DEC, NUMERIC, MONEY	:24<INT_DEC>	Field filled with *'s
BYTE, TEXT, DATETIME, INTERVAL	Not yet supplied	

G.6.2 Internal Decimal Fields

See *Appendix E Informix C-ISAM*.

G.7 Examples

Two examples demonstrate how FILETAB programs can be used to access an Informix database. The database used is the Informix sample `stores5` database.

- *Example 1 on page 8*
  Generate unpaid orders listing; selects all fields from the orders table; ignores orders with paid dates at `DETAB RECORD`; start positions in the `*DICT` are specified. Use of indicator variables to detect `NULL`s.
- *Example 2 on page 9*
  Selects all fields from the customer table; start positions in the `*DICT` are not specified as `*OPT ALI` is used to align fields. Use of indicator variable to detect `NULL`.

Note: The SQL statement is used to 'read' from the database into the FILETAB dictionary area. The names of the FILETAB fields need not be the same as the column names in the table.

For example, we are going to retrieve `customer_num` from customers as `custno`. It is not necessary to retrieve all the fields from a table. Specify the fields to be retrieved in both SQL and in

the FILETAB dictionary with the order of items in the FILETAB dictionary matching the order of fields specified in the SQL, that is

```
custno 0:4 [select customer_num, company
company 4/20 [from customers
or
company 0/20 [select company, customer_num
custno   20:4 [from customers
```

are both valid with `RL=24` in both cases.

G.7.1 Example 1

```
*PRO Example1
*FILE bodb5                     [local name
       OR=Informix              [database type
       NAME=bodb5               [database name
       RL=120                   [number of bytes returned
       SQL=sql1                 [name of *SQL
       IND=inds                 [name of *IND
       ONERROR=ErrorInSQL       [name of error *DETAB
*DIC
orderno         0:4             [integer
order_date      4:4<DATE>       [DATE
custno          8:4             [integer
ship_instruct  12/40            [char(40)
backlog        52/1             [char(1)
po_num         53/10            [char(10)
ship_date      64:4<DATE>       [DATE
ship_weight    68:24<INT_DEC>   [DECIMAL(8,2)
ship_charge    92:24<INT_DEC>   [MONEY
paid_date     116:4<DATE>       [DATE
pdind        -100:2             [NULL indicator for paid_date
sdind        -102:2             [NULL indicator for ship_date
siind        -104:2             [NULL indicator for ship_instruct
*
*IND inds                       [test for NULLs
paid_date :      pdind
ship_date :      sdind
ship_instruct :  siind

*SQL sql1
           select * from orders
*INL
C custno, A orderno , B order_date
D ship_instruct , E backlog
F po_num , G ship_date , H ship_weight
I ship_charge , J paid_date

*HEAD L CH0,1
```

```
                        example1: Unpaid Orders List



Order Order        Customer Back PO         Ship       Ship     Ship

Num   Date         Number   Log  Num        Date       Weight   Charge
*OUT L 1,1
AAAAA BBBBBBBBBBBB CCCCCC   E    FFFFFFFFFF GGGGGGGGGG HHHH.HH  IIII.II

                                <DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD>
*DET ErrorInSQL              [Error *DETAB
I DISPLAY SQLERRM/70
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1

*DETAB RECORD                    [For each record, check the indicators
C pdind LT 0       Y  -  N       [paid_date NULL?
  sdind LT 0       N  Y  -       [ship_date NULL?
A CALL CHECKSI     X  .  .
  IGNORE           .  X  X       [IGNORE if set

*DETAB CHECKSI
C siind LT 0        Y  N          [ship_instruct NULL?
A ship_instruct S   X  .          [space fill it

*GO
```

G.7.2 Example 2

```
*PRO Example2
*FILE bodb5                     [local name
       OR=Informix              [database type
       NAME=bodb5               [database name
       RL=134                   [number of bytes returned
       SQL=sql1                 [name of *SQL
       IND=inds                 [name of *IND
       ONERROR=ErrorInSQL       [name of error *DETAB
*
*DIC
custno       0:4                [serial(101)
fname         /15               [char(15)
lname         /15               [char(15)
company       /20               [char(15)
address1      /20               [char(20)
address2      /20               [char(20)
city          /15               [char(15)
state         /2                [char(2)
zipcode       /5                [char(5)
```

```
phone          /18               [char(18)
a2ind     -100:2                 [NULL Indicator field
*
*IND inds                        [Test for NULL address2
address2 :    a2ind
*
*SQL sql1
    select *
           from customer
*INL
A  custno
B  fname
C  lname , D company , E address1 , F address2 , G city
H  state , I zipcode , J phone
*
*HEAD L CH1,1
                   example2: Customer List
Cust No  Name and Address                      Telephone No

*OUT L 1,1
AAAAA    BBBBBBBBBBBBBBB CCCCCCCCCCCCCC        JJJJJJJJJJJJJJJJJJ

         DDDDDDDDDDDDDDDDDDDD

         EEEEEEEEEEEEEEEEEEEE

         FFFFFFFFFFFFFFFFFFFF

         GGGGGGGGGGGGGGG HH IIIII
*DET ErrorInSQL                  [Error *DETAB
I DISPLAY SQLERRM/70
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1
*
*DETAB RECORD                    [If Address2 is NULL (unprintable)
C a2ind LT 0    Y  N             [a2ind is negative
A address2 S    X  .             [space fill
*
*SORT *A                         [Sort on Customer no
*GO
```

Appendix H
INGRES Guide

H.1 About your INGRES Documentation

The *INGRES Guide* provides comprehensive details of how to access data held in an INGRES database and is divided into the following section:

H.1 About your INGRES Documentation	Describes what is available in this document and how it relates to the FILETAB software
H.2 Keywords and INGRES Files	Explains the keywords available with INGRES and how to use them
H.3 Directives and INGRES Files	Explains the directives available with INGRES and how to use them
H.4 Verbs and INGRES Files	Explains the verbs available with INGRES and how to use them
H.5 Reserved Words and INGRES Files	Explains the reserved words available with INGRES and how to use them
H.6 Field Types Supported with INGRES	Explains the field types supported with INGRES and how to use them
H.7 Examples	Demonstrates how to use INGRES

H.2 Keywords and INGRES Files

INGRES databases may be accessed by `*FILE` or `*?FILE` directives which have an `INGRES` organisation.

To access such files the SQL which will be used to extract information must be specified in a `*SQL` directive. This must be associated with the file by use of a keyword.

Additionally indicator variables can be specified in an `*INDICATORS` directive and associated with the file via a keyword.

A `*DETAB` for handling errors may also be specified via a keyword.

H.2.1 Organisation

Keyword	**OR**
Value	INGRES

In order to enable the INGRES organisation, the following UNIX or Linux environment variable should be set. Depending on the shell, enter the following:

In case of csh	setenv FTC_FLAGS “-DINGRES”
In case of ksh	FTC_FLAGS=“-DINGRES”; export FTC_FLAGS

H.2.2 SQL Directive Name

Keyword	**SQL** or **CURSOR**
Value	sql_directive_name

This will associate the *SQL directive sql_directive_name with the *?FILE name. The SQL in the named directive will be used to determine the view of the database on which File Handling verbs (and Fixed Logic) operate. The named SQL directive should only contain a single SELECT statement.

H.2.3 Record Length

Keyword	**RL**
Value	a positive integer
Default	80

This specifies the size of the data area returned by the SQL directive above. The size of the area must match the record length **exactly**. When calculating the data area size the following points should be considered:

- Columns defined as INTEGER1 occupy one byte
- Columns defined as DATE occupy twenty five bytes
- Columns defined as SMALLINT or INTEGER2 occupy two bytes and are aligned to a 2 byte boundary
- Columns defined as INTEGER4 occupy four bytes and are aligned to a 4 byte boundary
- Columns defined as FLOAT occupy eight bytes and are aligned to a 4 byte boundary
- Columns defined as CHARACTER will occupy their length

See below for details of the FILETAB field types which should be used for INGRES column types, *OPT ALI can be used to align fields in a *DICTIONARY automatically.

H.2.4 INDICATORS Directive Name

Keyword	**IND**
Value	`ind_directive_name`

This associates the `*INDICATORS` directive `ind_directive_name` with the `*?FILE` name. The `*INDICATORS` directive will be used to determine the indicator variables for columns read from the (database) file.

H.2.5 Error Handling *DETAB

Keyword	**ONERROR**
Value	`detab_name`

This specifies a `*DETAB` which will be called whenever an `error` or `warning` condition occurs whilst processing the file.

H.2.6 Database

Keyword	**NAME**
Value	`database_name`

This will specify the database name. The database name can also be specified by the usual file naming methods.

H.3 Directives and INGRES Files

H.3.1 *SQL Directive

Syntax:

```
*SQL name
SQL statements
```

This allows SQL statements to be specified, these can be used to retrieve columns from a database via a `READ` verb or by the fixed logic. It can also be used to specify SQL statements which can be executed using an `EXECSQL` verb.

FILETAB Host Fields can be included in the SQL statements. A Host Field will consist of a **:** followed by a FILETAB field name.

Host Fields allow you to specify FILETAB fields whose contents will be substituted into the SQL (when it is executed) as SQL literal values. This allows you to specify selection criteria, and so on, at run time. FILETAB

/ fields will be treated as string literals, :, ;, $ and other numeric fields as numeric literals, date fields as (:4<DATE>).

Optionally a second : and FILETAB field name can be placed after the first; the second field will act as an indicator variable.

Indicator variables are used to indicate if a field contains a NULL value. To indicate a NULL value the appropriate indicator variable should be set to less than zero.

11.18.1 *INDICATORS Directive

Syntax:

```
*INDICATORS name
{Indicator_Definition}
```

where:

```
Indicator_Definition
```

is:

```
Column_Name : FILETAB_Field_Name
```

This specifies FILETAB fields, which will be used as indicator variables for columns retrieved using the READ verb or via the fixed logic. The Column_Name must conform to INGRES naming conventions. The FILETAB fields should be :2 fields. The *INDICATORS directive is associated with a file using the IND Keyword.

When data is retrieved if a named column contains a NULL value its indicator variable will be set to less than zero.

H.4 Verbs and INGRES Files

H.4.1 CONNECT file_name

This verb, when executed, will attempt to connect to the database given via the file_name.

H.4.2 DISCONNECT file_name

This verb, when executed, will attempt to disconnect from the database given via the file_name.

H.4.3 CLOSE file_name

This verb will close the file specified. The database will **not** be disconnected (this will happen at the end of the run unless a DISCONNECT is used).

H.4.4 READ file_name Field

This verb (which will also be called by the fixed logic) will retrieve the rows specified by the `*SQL` associated with the `file_name`, one at a time, into the area specified by the field.

Note: The file will be automatically opened and the database connected if required.

H.4.5 EXECSQL file_name sql_directive_name

This verb will execute the SQL in the `*SQL` directive `sql_directive_name` on the database specified by the file *`file_name`*. **EXECSQL will automatically load the database if required**. Any preparable SQL may be specified in the `*SQL` directive.

H.5 Reserved Words and INGRES Files

The following reserved words are available for obtaining information about the latest executed INGRES SQL:

Reserved Word	Explanation
SQLCODE:4	Is set to: - 0 if an SQL statement executes OK - Greater than 0 if the SQL statement executes OK but an exception has occurred - Value 100 if a SELECT retrieves nothing - Less than zero on an error
SQLERRM/70	Error message text
SQLERRD2:4	Number of rows processed
SQLWARN0/1	W if any of sqlwarn[1-6] reserved words contain W
SQLWARN1/1	W if any truncation occurred or database has transactions
SQLWARN3/1	W if number of columns in the select list is not equal to the number of items in the INTO of the SELECT
SQLWARN4/1	W if no WHERE clause on prepared UPDATE, DELETE

These reserved words are not available until after the first access of the database

H.6 Field Types Supported with INGRES

The FILETAB field types which correspond to INGRES SQL column types are:

INGRES SQL Type	FILETAB Field	Type
VARCHAR(m)	/m+1<STRING>	String
INTEGER1	:1	Integer
SMALLINT, INTEGER2	:2	Integer
INTEGER, INTEGER4	:4	Integer
DATE	/25	Character
FLOAT	$8	Floating point

H.6.1 How FILETAB prints INGRES NULL fields

INGRES SQL Type	FILETAB Field	Print Values
VARCHAR (m)	/m+1<STRING>	A blank field
INTEGER1	:1	0
SMALLINT, INTEGER2	:2	0
INTEGER, INTEGER4	:4	0
DATE	/25	A blank field
FLOAT	$8	0

H.7 Examples

Three examples demonstrate how FILETAB programs can be used to access an INGRES database. The database used is the INGRES sample personnel database.

- *Example 1 on page 7*
  Selects all fields from the emp table; start positions in the *DICT are specified. Use of <STRING> for VCHAR data.
- *Example 2 on page 8*
  Selects all fields from the emp table; start positions in the *DICT are not specified as *OPT ALI is used to align fields. Use of indicator variable to detect NULL.
- *Example 3 on page 9*
  Selects data from three tables.

Note: The SQL statement is used to 'read' from the database into the FILETAB dictionary area.

The names of the FILETAB fields need not be the same as the

column names in the table, for example, we are going to retrieve `name` *from* `emp` *as* `ename` *and* `div` *from* `emp` *as* `ediv`.

It is not necessary to retrieve all the fields from a table. Specify the fields to be retrieved in both SQL and in the FILETAB dictionary with the order of items in the FILETAB dictionary matching the order of fields specified in the SQL, that is:

```
ename     0/11<STRING>     [select name, div
ediv      11/4<STRING>     [from emp
```

or

```
ediv      0/4<STRING>      [select div, name
ename     4/11<STRING>     [from emp
```

are both valid with RL=15 in both cases.

H.7.1 Example 1

```
*PRO Example1
*OPT DP=2                    [two places of decimal
*
*ALTER FSCT5 [FSC's for floating point
01
*
*FILE bod5 [local name
       OR=Oracle  [database type
       NAME=personnel         [database name
       RL=60 [number of bytes returned
       SQL=sql1               [name of *SQL
USER=ftusr/ftusr99 [username and password
       ONERROR=ErrorInSQL     [name of error *DETAB
*
*SQL sql1
           select * from emp
*
*DIC
empno=0:4[Number (4)
ename      =  4/10 [Varchar2(10)
job =  14/9 [Varchar2(9)
mgr =  24/4 [Number(4)
hiredate=  28/10 [Date
sal =  40$8 [Number (7,2)
comm =  48$8 [Number (7,2)
deptno = 56:4 [Number (2)
*
*HEAD L CH1, 1
example1: Employee data
```

```
*
*INL
A ename                       [vchar(10)
0 esalary                     [float
B edept                       [vchar(8)
C ediv                        [vchar(3)
D emgr                        [vchar(10)
E ebirthdate                  [date
1 enumdep                     [integer2
*
*HEAD L CH1,1                 [Heading
                example1: Employee Listing

Name              Salary Dept      Div  Manager     Birth Date Dep 1
*OUT L 1,0                    [Detail line
AAAAAAAAAAA $*0000000.00 BBBBBBBBB CCCC DDDDDDDDDDD EEEEEEEEEEE 11

*DET ErrorInSQL               [Error *DETAB
I DISPLAY SQLERRM/70
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1
*GO
```

H.7.2 Example 2

```
*PRO Example2
*OPT DP=2                    [No of decimal places
*                            [when moving floating field
*                            [to a character field
*
*OPT ALI                     [Align *DIC entries
*                            [$8 Fields align onto 4 byte boundary
*                            [:2 Fields align onto 2 byte boundary
*
*FILE  emp1                  [local name
       OR=Ingres             [database type
       NAME=personnel        [database name
       RL=72                 [number of bytes returned
       SQL=sql1              [name of *SQL
       IND=inds1             [name of *IND
       ONERROR=ErrorInSQL    [name of error *DETAB
*
*SQL sql1
              select *from emp
*
*DIC
ename      =   0/11<STRING>  [vchar(10)
esalary    =    $8           [money - Floating Point
edept      =    /9<STRING>   [vchar(8)
```

```
ediv       =    /4<STRING>   [vchar(3)
emgr       =    /11<STRING>  [vchar(10)
ebirthdate =    /25          [date
enumdep    =    :2           [integer2 - 2 byte binary
Ind1       = 500:2           [Used to test for NULL mgr
*
*IND inds1                   [Test for null mgr
mgr : Ind1
*
*ALTER FSCT5                 [FSC to be used for the float field
0
*INL
A ename[vchar(10)
0 esalary[float
B edept                      [vchar(8)
C ediv                       [vchar(3)
D emgr                       [vchar(10)
E ebirthdate                 [date
1 enumdep                    [integer2
*
*HEAD L CH1,1                [Heading
                example2: Employee Listing II

Name              Salary Dept      Div  Manager     Birth Date Dep 1
*OUT L 1,0                   [Detail line
AAAAAAAAAAA $*0000000.00 BBBBBBBBB CCCC DDDDDDDDDDD EEEEEEEEEEE 11

*DETAB  RECORD
C Ind1 LT 0                Y  N  [Is mgr field NULL?
A emgr/10 MV '**********'  X  .  [Fill vchar(10)
*DET ErrorInSQL              [Error *DETAB
I DISPLAY SQLERRM/70
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1
*GO
```

H.7.3 Example 3

```
*PRO Example3
*
*FILE  emp1                     [local name
       OR=Ingres                [database type
       NAME=personnel           [database name
       RL=49                    [number of bytes returned
       SQL=sql1                 [name of *SQL
       ONERROR=ErrorInSQL       [name of error *DETAB
*
*SQL sql1
    select emp.name ,
```

```
                    dept.dname ,
                    dept.div   ,
                               bldg.city  ,
                               bldg.state ,
                               bldg.zip
       from emp , dept , bldg
       where emp.dept = dept.dname and
             emp.div  = dept.div   and
            dept.bldg = bldg.bldg
*
*DIC
ename      =   0/11<STRING>     [vchar(10)
ddept      =  11/9<STRING>      [vchar(8)
ddiv       =  20/4<STRING>      [vchar(3)
bcity      =  24/16<STRING>     [vchar(15)
bstate     =  40/3<STRING>      [vchar(2)
bzip       =  43/6<STRING>      [vchar(5)
*INL
A ename                         [vchar(10)
B ddept                         [vchar(8)
C ddiv                          [vchar(3)
D bcity                         [vchar(15)
E bstate                        [vchar(2)
F bzip                          [vchar(5)
*
*HEAD L CH1,1                   [Heading
example3: Employee Location

Name        Dept      Div  City             St  Zip
*OUT L     1,0                     [Detail line
AAAAAAAAAAA BBBBBBBBB CCCC DDDDDDDDDDDDDDDD EEE FFFFFF

*DET ErrorInSQL                 [Error *DETAB
I DISPLAY SQLERRM/70
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1
*GO
```

Appendix I
ORACLE Guide

I.1 About your ORACLE Guide

The *ORACLE Guide* provides comprehensive details of how to access data held in an ORACLE database and is divided into the following sections:

I.1 About your ORACLE Guide	Describes what is available in this document and how it relates to the FILETAB software
I.2 Keywords and ORACLE Files	Explains the keywords available with ORACLE and how to use them
I.3 Directives and ORACLE Files	Explains the directives available with ORACLE and how to use them
I.4 Verbs and ORACLE Files	Explains the verbs available with ORACLE and how to use them
I.5 Reserved Words and ORACLE Files	Explains the reserved words available with ORACLE and how to use them
I.6 Field Types Supported with ORACLE	Explains the field types supported with ORACLE and how to use them
I.7 Examples	Demonstrates how to use ORACLE

I.2 Keywords and ORACLE Files

ORACLE databases may be accessed by `*FILE` or `*?FILE` directives which have an `ORACLE` organisation.

To access such files the SQL which will be used to extract information must be specified in a `*SQL` directive. This must be associated with the file by use of a keyword.

Additionally indicator variables can be specified in an `*INDICATORS` directive and associated with the file via a keyword.

A `*DETAB` for handling errors may also be specified via a keyword.

I.2.1 Organisation

Keyword	**OR**
Value	ORACLE

In order to enable the ORACLE organisation, the following UNIX or Linux environment variable should be set. Depending on the shell, enter the following:

In case of csh	setenv FTC_FLAGS "-DINGRES"
In case of ksh	FTC_FLAGS="-DINGRES"; export FTC_FLAGS

I.2.2 SQL Directive Name

Keyword	**SQL** or **CURSOR**
Value	sql_directive_name

This will associate the *SQL directive sql_directive_name with the *?FILE name. The SQL in the named directive will be used to determine the view of the database on which File Handling verbs (and Fixed Logic) operate. The named SQL directive should only contain a single SELECT statement.

I.2.3 Record Length

Keyword	**RL**
Value	a positive integer
Default	80

This specifies the size of the data area returned by the SQL directive above. The size of the area must match the record length **exactly**. When calculating the data area size the following points should be considered:

- Columns defined as DATE occupy ten bytes
- Columns defined as NUMBER(p) occupy four bytes (usually, size of int)
- Columns defined as FLOAT or NUMBER(p,s) occupy eight bytes
- Columns defined as CHARACTER will occupy their length

See below for details of the FILETAB field types which should be used for ORACLE column types. (*OPT ALI can be used to align fields in a *DICTIONARY automatically).

I.2.4 INDICATORS Directive Name

Keyword	**IND**
Value	ind_directive_name

This associates the *INDICATORS directive ind_directive_name with the *?FILE name. The *INDICATORS directive will be used to determine the indicator variables for columns read from the (database) file.

I.2.5 Error Handling *DETAB

Keyword	**ONERROR**
Value	detab_name

This specifies a *DETAB which will be called whenever an error or warning condition occurs whilst processing the file.

I.2.6 Database

Keyword	**NAME**
Value	database_name

This will specify the database name. The database name can also be specified by the usual file naming methods. The database name usually maps onto a valid ORACLE SID name.

I.2.7 User

Keyword	**USER**
Value	username/password

This will specify the user name which is used to connect to the database. This keyword is optional if the environment variable FTC_ORACLE_USER is specified.

I.3 Directives and ORACLE Files

I.3.1 *SQL Directive

Syntax:

```
*SQL name
SQL statements
```

This allows SQL statements to be specified which can be used to retrieve columns from a database via a READ verb or by the fixed logic. It can also be used to specify SQL statements which can be executed using an EXECSQL verb.

Included in the SQL statements can be FILETAB Host Fields. A Host Field will consist of a : followed by a FILETAB field name.

Host Fields allow you to specify FILETAB fields whose contents will be substituted into the SQL (when it is executed) as SQL literal values. This allows you to specify selection criteria, and so on, at run time. FILETAB / fields will be treated as string literals, :, ;, $ and other numeric fields as numeric literals, date fields as (:4<DATE>).

Indicator variables are used to indicate if a field contains a NULL value. To indicate a NULL value the appropriate indicator variable should be set to less than zero.

I.3.2 *INDICATORS Directive

Syntax:

```
*INDICATORS name
{Indicator_Definition}
```

where:

```
Indicator_Definition
```

is:

```
Column_Name : FILETAB_Field_Name
```

This directive is used to specify FILETAB fields, which will be used as indicator variables for columns retrieved using the READ verb or via the fixed logic. The Column_Name must conform to ORACLE naming conventions. The FILETAB fields should be :2 fields. The *INDICATORS directive is associated with a file using the IND keyword.

When data is retrieved, if a named column contains a NULL value its indicator variable will be set to less than zero.

I.4 Verbs and ORACLE Files

I.4.1 CONNECT file_name

This verb, when executed, will attempt to connect to the database given via the file_name.

I.4.2 DISCONNECT file_name

This verb, when executed, will attempt to disconnect from the database given via the file_name.

I.4.3 CLOSE file_name

This verb will close the file specified. The database will **not** be disconnected from (this will happen at the end of the run unless a `DISCONNECT` is used).

I.4.4 READ file_name Field

This verb (which will also be called by the fixed logic) will retrieve the rows specified by the `*SQL` associated with the `file_name`, one at a time, into the area specified by the field.

The file will be automatically opened and the database connected to if required.

I.4.5 EXECSQL file_name sql_directive_name

This verb will execute the SQL in the `*SQL` directive `sql_directive_name` on the database specified by the file `file_name`. **`EXECSQL` will automatically load the database if required**. Any preparable SQL may be specified in the `*SQL` directive.

I.5 Reserved Words and ORACLE Files

The following reserved words are available for obtaining information about the latest executed ORACLE SQL:

Reserved Word	Explanation
`SQLCODE:4`	is set to `0` if an SQL statement executes `OK`, greater than 0 if the SQL statement executes `OK` but an exception has occurred, value 1403 if a `SELECT` retrieves nothing, or less than zero on an error
`SQLERRM/72`	error message parameters
`SQLERRD2:4`	number of rows processed
`SQLWARN0/1`	`W` if any of `sqlwarn[1-5]` reserved words contain `W`
`SQLWARN1/1`	`W` if any truncation occurred or database has transactions
`SQLWARN3/1`	`W` if number of columns in the select list is not equal to the `number` of host variables in the `INTO` of the `SELECT`
`SQLWARN4/1`	`W` if no `WHERE` clause on prepared `UPDATE, DELETE`

These reserved words are not available until after the first access of the database.

I.6 Field Types Supported with ORACLE

The FILETAB field types which correspond to ORACLE SQL column types are:

ORACLE SQL Type	FILETAB Field	Type
VARCHAR(n), CHAR(n)	/n	String
DATE	/10	String
FLOAT, NUMBER(p,s)	$8	Floating point
ROWID	/18	String
NUMBER(p)	:4	Integer
RAW, LONG RAW,		Not yet supported

I.6.1 How FILETAB Points ORACLE NULL Fields

ORACLE SQL Type	FILETAB Field	Print Values
VARCHAR (n), CHAR(n)	/n	A blank field
DATE	/10	A blank field
FLOAT, NUMBER(p,s)	$8	0
ROWID	/18	A blank field
NUMBER(p)	:4	0
RAW, LONG RAW	Not yet supported	

I.7 Examples

This section contains examples which show how FILETAB programs can be used to access an ORACLE database. The examples use two tables (`emp` and `dept`) from the sample ORACLE database (see your ORACLE Guide).

- *Example 1 on page 7*
  Selects all fields from the `emp` table: start positions in the `*DICT` are specified.
- *Example 2 on page 8*
  Selects all fields from the `emp` table; start positions in the `*DICT` are not specified as `*OPT ALI` is used to align fields.
- *Example 3 on page 9*
  Selects data from two tables.

Note: The SQL statement is used to 'read' from the database into the FILETAB dictionary area.

The names of the FILETAB fields need not be the same as the column names in the table. For example, we are going to retrieve `name` *from* `emp` *as* `ename`.

It is not necessary to retrieve all the fields from a table. Specify the fields to be retrieved in both SQL and in the FILETAB dictionary with the order of items in the FILETAB dictionary matching the order of fields specified in the SQL, that is:

```
job        0/9          [select job, name
ename      9/10         [from emp
```

or

```
ename      0/10         [select name, job
job        10/9         [from emp
```

are both valid with `RL=19` *in both cases.*

I.7.1 Example 1

```
*PRO Example1
*OPT DP=2                       [No of decimal places
*                               [when moving floating field
*                               [to a character field
*
*FILE  emp1                     [local name
       OR=Ingres                [database type
       NAME=personnel           [database name
       RL=72                    [number of bytes returned
       SQL=sql1                 [name of *SQL
       ONERROR=ErrorInSQL       [name of error *DETAB
*
*SQL sql1
           select * from emp
*
*DIC
ename      =  0/11<STRING>      [vchar(10)
esalary    =  12$8              [money - Floating Point
edept      =  20/9<STRING>      [vchar(8)
ediv       =  29/4<STRING>      [vchar(3)
emgr       =  33/11<STRING>     [vchar(10)
ebirthdate =  44/25             [date
enumdep    =  70:2              [integer2 - 2 byte binary
*
*
*ALTER FSCT5                    [FSC to be used for the float field
0
*
```

```
*INL
A ename                         [vchar(10)
0 esalary                       [float
B edept                         [vchar(8)
C ediv                          [vchar(3)
D emgr                          [vchar(10)
E ebirthdate                    [date
1 enumdep                       [integer2
*
*HEAD L CH1,1                   [Heading
                example1: Employee Listing

Name              Salary Dept      Div  Manager     Birth Date Dep 1
*OUT L 1,0                      [Detail line
AAAAAAAAAAA $*0000000.00 BBBBBBBBB CCCC DDDDDDDDDDD EEEEEEEEEEE 11


*DET ErrorInSQL                 [Error *DETAB
I DISPLAY SQLERRM/70
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1
*GO
```

I.7.2 Example 2

```
*PRO Example2
*OPT DP=2                 [two places of decimal
*
*OPT ALI                  [Align database area
*
*ALTER FSCT5              [FSC's for floating point
01
*
*FILE bodb5                  [local name
       OR=Oracle             [database type
       NAME=personnel        [database name
       RL=60                 [number of bytes returned
       SQL=sql1              [name of *SQL
       USER=ftusr/ftusr99    [username and password
       ONERROR=ErrorInSQL    [name of error *DETAB
*
*SQL sql1
    select *
           from emp
*DIC
empno         =  0:4       [Number(4)
ename         =   /10      [Varchar2(10)
job           =   /9       [Varchar2(9)
mgr           =   :4       [Number(4)
```

```
hiredate       =   /10      [Date
sal            =   $8       [Number(7,2)
comm           =   $8       [Number(7,2)
deptno         =   :4       [Number(2)
*
*INL
A empno , B ename , C job , D mgr , E hiredate , 0 sal
1 comm  , F deptno
*
*HEAD L CH1,1
                   example2: Employee listing II

No   Name       Job       Mgr  Hired          Salary  Commission Dept No
*OUT L    1,1          [1234567890123456789012345
AAAA BBBBBBBBBB CCCCCCCCC DDDD EEEEEEEEEE $*000000.00 $*111111.11 FF
*DET ErrorInSQL
I DISPLAY SQLERRM/70
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1
*GO
```

I.7.3 Example 3

```
*PRO Example3
*FILE bodb5                      [local name
       OR=Oracle                 [database type
       NAME=personnel            [database name
       RL=55                     [number of bytes returned
       SQL=sql1                  [*SQL name
       USER=ftusr/ftusr99        [username and password
       ONERROR=ErrorInSQL        [name of error *DETAB
*
*SQL sql1
    select emp.empno , emp.ename , emp.job , emp.deptno ,
           dept.dname , dept.loc
       from emp  , dept
       where emp.deptno = dept.deptno
*DIC
empno          =  0:4       [Number(4)
ename          =  4/10      [Varchar2(10)
job            = 14/9       [Varchar2(9)
deptno         = 24:4       [Number(2)
ddept          = 28/14      [Varchar2(14)
loc            = 42/13      [Varchar2(13)
*INL
A empno , B ename , C job , D deptno ,
E ddept , F loc
*HEAD L CH1,1
              example3: Employee Location
```

```
No   Name       Job       Dept Dept name      Location
*OUT L  1,1         [1234567890123456789012345
AAAA BBBBBBBBBB CCCCCCCCC DDDD EEEEEEEEEEEEEE FFFFFFFFFFFF
*DET ErrorInSQL
I DISPLAY SQLERRM/70       [error *DETAB
  EXECUTE printf('SQL Error %ld%c', SQLCODE, 10)
  STOP 1
*SORT *A                   [sort by empno
*GO
```

Appendix J
HTMLX Guide

J.1 Overview

HTMLX allows FILETAB to produce enhanced web pages. Some examples of the types of pages that can be produced are:

- Automatic HTML table based versions of FILETAB output utilising fonts, colours, formats, page links etc.
- Non-table based pages e.g. graphic reports, data based forms, XML/XSL output.
- Sophisticated Javascript (or VBScript) applications e.g. report outliners, complex tree navigators.

J.1.1 About Your HTMLX Documentation

The *HTMLX Guide* provides comprehensive details of how to use HTMLX to produce web pages and is divided into the following sections:

J.1 Overview	Describes what is available in this document and how it relates to the FILETAB software
J.2 Options and HTMLX	Explains the keywords available with HTMLX and how to use them
J.3 Reserved Words and HTMLX	Explains the reserved words available with HTMLX and how to use them
J.4 HTML Decision Point and HTMLX	Explains the use of `*DETAB HTML` available with HTMLX
J.5 Mapline and Zone Information Generation	Explains the generation of the mapline and zone information
J.6 Examples	Demonstrates how to use HTMLX

J.2 Options and HTMLX

HTMLX is controlled by the following `*OPTION`s:

`HTMLX`	Enables HTMLX output
`HTMLXC`	Enables HTMLX CGI input

HTMLXD	Adds BORDER clause to the HTML <TABLE> element – this shows the table structure more clearly and can be an aid when calculating zone numbers
HTMLXO	By default, *HEAD items are translated to <TH> elements and *OUT items to <TD> elements (allowing for their differing rendering, for example, bold and centred). This option disables this translation and converts all output items to <TD> elements
HTMLXS	By default, output zones are automatically positioned by the rendering of the HTML <TABLE> element (according to the table cell contents). This option applies a more rigorous spacing construct to the table cells. When combined with the HTMLXO option above produces output closest to the original FILETAB report

J.3 Reserved Words and HTMLX

The following reserved words are available at *DETAB HTML when amending the automatic HTML table based output:

Name	Type	Use
HTML-TYPE	/1	Holds the hook type when calling *DETAB HTML: **O**pen, called when opening output **C**lose, called when closing output. **P**rint, called when producing output **Z**ero, called if no records selected.
HTML-MAP	/1	Holds the type of zone: **B**lank, spacefilled zone **L**eft, left aligned zone **R**ight, right aligned zone **D**efault, other zone.
HTML-PPN	:4	Holds the current zone number: **Z**one 0, start of table row **Z**one n, table cell n.
HTML-PPA	:4	Pointer to address of PPn for zone
HTML-PPS	:4	Start position of PPn for zone
HTML-PPL	:4	Length of zone
HTML-PREFIX	:4	Pointer to HTML prefix null terminated string

HTML-DATA	:4	Pointer to HTML data null terminated string
HTML-POSTFIX	:4	Pointer to HTML postfix null terminated string

J.4 HTML Decision Point and HTMLX

If *DETAB HTML is specified it is called each time an HTMLX hook is available. It is up to the programmer to decode the hook type and amend the automatic HTML table based output as required. It should be noted that *DETAB HTML is called after *DETAB PRINT if it is also specified.

During default processing the contents of the HTML prefix, data and postfix strings are appended to the output and these can be amended or replaced as required during each call to the hook. However, various other options are possible at each of the hooks.

J.4.1 Open Hook

This hook is called when opening the HTML output and the contents of the reserved words are as follows:

Name	Contents
HTML-TYPE	**O**, open hook
HTML-MAP	Spacefilled
HTML-PPN	Zeroised
HTML-PPA	A'PP1
HTML-PPS	1
HTML-PPL	Print line length, for example, 160
HTML-PREFIX	<HTML><HEAD></HEAD><BODY> statements
HTML-DATA	Null
HTML-POSTFIX	<TABLE> statement

J.4.1.1 Exit Options

If control is returned with EXIT T (implicitly or explicitly) then default processing takes place.

If control is returned by EXIT F (or IGNORE) or the postfix string is set to null then the automatic HTML table based output is disabled and a simpler format is produced.

J.4.1.2 Amending the HTML Data String

Setting the data string to a valid filename will replace the automatic HTML statements with the contents of the file.

J.4.2 Close Hook

This hook is called when closing the HTML output and the contents of the reserved words are as follows:

Name	Contents
HTML-TYPE	**C**, close hook
HTML-MAP	Spacefilled
HTML-PPN	Zeroised
HTML-PPA	A'PP1
HTML-PPS	1
HTML-PPL	Print line length, for example, 160
HTML-PREFIX	</TABLE> statement, table based or Null, non-table based
HTML-DATA	Null
HTML-POSTFIX	</BODY></HTML> statement

J.4.2.1 Exit Options

If control is returned with `EXIT T` (implicitly or explicitly) then default processing takes place.

If control is returned by `EXIT F` (or `IGNORE)` then the automatic HTML statements are appended to the output.

J.4.2.2 Amending the HTML Data String

Setting the data string to a valid filename will replace the automatic HTML statements with the contents of the file.

J.4.3 Print hook : Zone 0 - Start of Table Row

This hook is called when producing the HTML output at the start of a table row and the contents of the reserved words are as follows:

Name	Contents
HTML-TYPE	**P**, print hook
HTML-MAP	Spacefilled

HTML-PPN	Zeroised
HTML-PPA	A'PP1
HTML-PPS	1
HTML-PPL	Print line length, for example, 160
HTML-PREFIX	<TR> statement, table based or Null, non-table based
HTML-DATA	Contents of full mapline
HTML-POSTFIX	</TR> statement, table based or statement, non-table based

J.4.3.1 Exit Options

If control is returned with EXIT T (implicitly or explicitly) then default processing takes place.

If control is returned by EXIT F (or IGNORE) then the automatic HTML statements are appended to the output.

J.4.3.2 Amending the HTML Data String

Amending the HTML data string will alter the generated mapline and therefore affect the zone information.

J.4.4 Print hook : Zone n - Table Cell

This hook is called when producing the HTML output for each table cell and the contents of the reserved words are as follows:

Name	Contents
HTML-TYPE	**P**, print hook
HTML-MAP	Type of zone
HTML-PPN	Zone number (1 – n)
HTML-PPA	A'PPn
HTML-PPS	N
HTML-PPL	Zone length
HTML-PREFIX	<TH> or <TD> statement, table based or Null, non-table based
HTML-DATA	^

HTML-POSTFIX	</TH> or </TD> statement, table based or statement, non-table based

J.4.4.1 Exit Options

If control is returned with EXIT T (implicitly or explicitly) then default processing takes place.

If control is returned by EXIT F (or IGNORE) then the automatic HTML statements are appended to the output.

J.4.4.2 Amending the HTML Data String

When appending the HTML data string '^'s will be replaced by the contents of the zone.

J.4.5 Zero Hook

This hook is called when producing the HTML output and no records have been selected. Its behaviour is similar to the print hook and passes a single zone containing 'NO RECORDS SELECTED'.

J.5 Mapline and Zone Information Generation

When a print line is ready to output a mapline is generated by analysis of the *INLIST and print line contents. This is then passed to *DETAB HTML (as zone 0) for amendment as required.

The mapline contains a character code corresponding to each print position (PPn) depending upon whether the print position is part of a **B**lank, **L**eft aligned, **R**ight aligned or **D**efault (other) zone.

To generate zone information the mapline is scanned and consecutive character codes of the same type form a zone

J.6 Examples

Two examples demonstrate how FILETAB programs can be used to produce HTML output. The examples use the ODBC interface for FILETAB to access an ODBC compliant database. The database consists of tables holding sample data for Sales Representatives, Customers, Orders, Order Lines and Stock Items.

- *Example 1 Static HTML Output on page 7*
  This program produces HTML output based on a date range input on the command line when the program is run. The output file is copied to the Web Server to give access to browser users.
- *Example 2 Dynamic HTML Output on page 8*
  This program produces HTML output based on a date range input by the browser user (on a HTML form for example). In this

scenario the executable program itself is copied to the Web Server and is run at the time of the request.

J.6.1 Example 1 Static HTML Output

```
*PROGRAM EXAMPLE1                        [START OF PROGRAM
*
*OPTION HTMLX,HTMLXO,HTMLXS              [ENABLE HTML OUTPUT
*
*OPTION PL=80,OC                         [PRINT LENGTH, OMIT COMMENTS
*
*FILE SALES                              [MAIN FILE LOCAL NAME
          OR=ODBC                        [DATABASE ORGANISATION
          NAME='FT Examples'             [ODBC DATASOURCE NAME (DSN)
          RL=95                          [RECORD LENGTH
          SQL=SALESQUERY                 [*SQL STATEMENT NAME
*
*IFILE ARGS                              [INPUT FILE LOCAL NAME
          OR=CLI                         [COMMAND LINE ARGUMENTS
          MAP=ARGSMAP                    [ARGUMENT *MAP NAME
*
*DICTIONARY                              [DATA DICTIONARY
  REGION      = /10                      [SALES REGION
  NAME        = /30                      [SALES REPRESENTATIVE
  ITEM        = /6                       [ITEM CODE
  DESCRIPTION = /30                      [ITEM DESCRIPTION
  SALES       = /19                      [SALES VALUE
*
  DATEARGS = /16                         [CLI ARGUMENTS INPUT
  FROMDATE = DATEARGS/8                  [MAP TO DATE FROM
  TODATE   = /8                          [AND DATE TO
  DATEFROM = :4<DATE>                    [CONVERTED DATE FROM
  DATETO   = :4<DATE>                    [AND DATE TO
  VALUE    = :8                          [CONVERTED SALES VALUE
  P1       = :4                          [POINTER
*
*INLIST                                  [INLIST AREA
  N REGION                               [GROUP BY SALES REGION
  M NAME                                 [AND THEN SALES REPRESENTATIVE
  A FROMDATE TODATE                      [DATE SELECTION RANGE
  B ITEM                                 [ITEM DETAILS
  C DESCRIPTION
  1 VALUE                                [SALES VALUE (TO BE TOTALED)
*
*HEAD L,N CH1,1                        [HEADING FOR EACH REGION (& PAGE)
Run on DD/MM/YYYY                                               Page PPPP

         'NNNNN'Regional Sales Analysis for'AAAAAAAA'to'AAAAAAAA'
```

```
 Sales Representative      Item/Description                      Sales
*
*HEAD M 1,0                            [HEADING FOR EACH SALES REP.
'MMMMMMMMMMMMMMMMMMMMMMMMMMM'
*
*OUT L 0,1                             [OUTPUT FOR EACH ITEM
                         BBBBBB/CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  11111.11
*
*OUT M 0,1                           [TOTAL OUTPUT FOR EACH SALES REP.
                         'All Items'                            11111.11
*
*OUT N 1,1                             [TOTAL OUTPUT FOR EACH REGION
 NNNNN'Regional Sales Total'                                   11111.11
*
*OUT F 1,0                             [GRAND TOTAL OUTPUT
'Regional Sales Grand Total'                                    11111.11
*
*MAP ARGSMAP                           [CLI ARGUMENTS MAP
  :FROMDATE
  :TODATE
*
*SQL SALESQUERY
SELECT Region, Sales.Name, Item, Description, Sum(Value) AS Sales
FROM (((Items INNER JOIN Lines ON Items.ID = Lines.Item)
              INNER JOIN Orders ON Lines.Order = Orders.ID)
              INNER JOIN Customers ON Orders.Customer = Customers.ID)
              INNER JOIN Sales ON Customers.Sales = Sales.ID
WHERE Ordered BETWEEN #:DATEFROM# AND #:DATETO#
GROUP BY Region, Sales.Name, Item, Description
*
*DETAB START                           [PROCESS AT START OF RUN
I READ ARGS DATEARGS/0                 [READ CLI ARGUMENTS
  DATEFROM MV FROMDATE                 [CONVERT DATE FROM
  DATETO MV TODATE                     [AND DATE TO
*
*DETAB RECORD                          [PROCESS FOR EACH RECORD
I VALUE MV SALES/14                    [CONVERT SALES VALUE POUNDS
  VALUE * 100                          [CONVERT TO PENCE
  P1 MV A'SALES                        [POINT TO START OF SALES VALUE
  P1 ST '.'                            [SKIP TO THE DECIMAL POINT
  P1 + 1                               [AND THEN OVER IT
  VALUE + (P1)/2                       [ADD IN THE SALES VALUE PENCE
*
*STOP                                  [END OF PROGRAM
```

J.6.2 Example 2 Dynamic HTML Output

```
*PROGRAM EXAMPLE2                      [START OF PROGRAM
*
```

```
*OPTION HTMLX,HTMLXC,HTMLXO,HTMLXS        [ENABLE HTML OUTPUT
*
*OPTION PL=80,OC                          [PRINT LENGTH, OMIT COMMENTS
*
*FILE SALES                               [MAIN FILE LOCAL NAME
          OR=ODBC                         [DATABASE ORGANISATION
          NAME='FT Examples'              [ODBC DATASOURCE NAME (DSN)
          RL=95                           [RECORD LENGTH
          SQL=SALESQUERY                  [*SQL STATEMENT NAME
*
*IFILE ARGS                               [INPUT FILE LOCAL NAME
          OR=CGI                          [COMMAND LINE ARGUMENTS
          MAP=ARGSMAP                     [ARGUMENT *MAP NAME
*
*DICTIONARY                               [DATA DICTIONARY
  REGION      = /10                       [SALES REGION
  NAME        = /30                       [SALES REPRESENTATIVE
  ITEM        = /6                        [ITEM CODE
  DESCRIPTION = /30                       [ITEM DESCRIPTION
  SALES       = /19                       [SALES VALUE
*
  DATEARGS = /16                          [CLI ARGUMENTS INPUT
  FROMDATE = DATEARGS/8                   [MAP TO DATE FROM
  TODATE   = /8                           [AND DATE TO
  DATEFROM = :4<DATE>                     [CONVERTED DATE FROM
  DATETO   = :4<DATE>                     [AND DATE TO
  VALUE    = :8                           [CONVERTED SALES VALUE
  P1       = :4                           [POINTER
*
*INLIST                                   [INLIST AREA
  N REGION                                [GROUP BY SALES REGION
  M NAME                                  [AND THEN SALES REPRESENTATIVE
  A FROMDATE TODATE                       [DATE SELECTION RANGE
  B ITEM                                  [ITEM DETAILS
  C DESCRIPTION
  1 VALUE                                 [SALES VALUE (TO BE TOTALED)
*
*HEAD L,N CH1,1                         [HEADING FOR EACH REGION (& PAGE)
Run on DD/MM/YYYY                                             Page PPPP

         'NNNNN'Regional Sales Analysis for'AAAAAAAA'to'AAAAAAAA'

 Sales Representative      Item/Description                       Sales
*
*HEAD M 1,0                               [HEADING FOR EACH SALES REP.
'MMMMMMMMMMMMMMMMMMMMMMMMMM'
*
*OUT L 0,1                                [OUTPUT FOR EACH ITEM
```

```
                         BBBBBB/CCCCCCCCCCCCCCCCCCCCCCCCCCCCC  11111.11
*
*OUT M 0,1                           [TOTAL OUTPUT FOR EACH SALES REP.
                         'All Items'                           11111.11
*
*OUT N 1,1                              [TOTAL OUTPUT FOR EACH REGION
 NNNNN'Regional Sales Total'                                  11111.11
*
*OUT F 1,0                              [GRAND TOTAL OUTPUT
'Regional Sales Grand Total'                                   11111.11
*
*MAP ARGSMAP                            [CLI ARGUMENTS MAP
  :FROMDATE
  :TODATE
*
*SQL SALESQUERY
SELECT Region, Sales.Name, Item, Description, Sum(Value) AS Sales
FROM (((Items INNER JOIN Lines ON Items.ID = Lines.Item)
              INNER JOIN Orders ON Lines.Order = Orders.ID)
              INNER JOIN Customers ON Orders.Customer = Customers.ID)
              INNER JOIN Sales ON Customers.Sales = Sales.ID
WHERE Ordered BETWEEN #:DATEFROM# AND #:DATETO#
GROUP BY Region, Sales.Name, Item, Description
*
*DETAB START                            [PROCESS AT START OF RUN
I READ ARGS DATEARGS/0                  [READ CLI ARGUMENTS
  DATEFROM MV FROMDATE                  [CONVERT DATE FROM
  DATETO MV TODATE                      [AND DATE TO
*
*DETAB RECORD                           [PROCESS FOR EACH RECORD
I VALUE MV SALES/14                     [CONVERT SALES VALUE POUNDS
  VALUE * 100                           [CONVERT TO PENCE
  P1 MV A'SALES                         [POINT TO START OF SALES VALUE
  P1 ST '.'                             [SKIP TO THE DECIMAL POINT
  P1 + 1                                [AND THEN OVER IT
  VALUE + (P1)/2                        [ADD IN THE SALES VALUE PENCE
*
*STOP                                   [END OF PROGRAM
```