FieldShield 4.1

Data Privacy Solutions

Software User Manual

FieldShield Version 4.1 **LEGAL NOTICE**

April 2016

INNOVATIVE ROUTINES INTERNATIONAL (IRI), INC.
2194 Highway A1A
Suite 303, Atlantis Center
Melbourne, Florida 32937 USA
Tel (321) 777-8889, Fax (321) 777-8886
Email fieldshield@iri.com
URL www.iri.com

Table of Contents

Introduction to FieldShield 9

1 What is FieldShield? 9
  1.1 FieldShield Features 9
  1.2 FieldShield User Recommendations 10

2 FieldShield Technical Specifications 11

3 FieldShield Package Contents 13

4 FieldShield Control Language (FCL) Purpose 14

5 Execution 15

6 Conventions 16
  6.1 Documentation Conventions 16
    6.1.1 UNIX / Linux or Windows comments 16
    6.1.2 Italics: Documentation Convention 16
  6.2 FCL Syntax Conventions 16
    6.2.1 Naming Conventions 16
    6.2.2 Abbreviations 17
    6.2.3 Optional Statement Parameters 17
    6.2.4 Environment Variables 17
    6.2.5 Line Continuation 18
    6.2.6 Comments 18

Static Data Masking Functions 19

1 Conditional Field Security and Masking 19
  1.1 Multiple IF THEN clauses 20
  1.2 Conditional Field and Data Statement Examples 21

2 Record Filtering (Omission) 24
  2.1 Syntax 24
  2.2 Include-Omit Statement Order Evaluation 24
  2.3 Include-Omit Examples 25
  2.4 Record Filtering 28
  2.5 Record Formatting 32

3 ASCII/ALPHANUM Encryption and Decryption 38
  3.1 Usage: ASCII Encryption and Decryption 39
  3.2 Syntax: ASCII/ALPHANUM Encryption 39

3.3 Syntax: ASCII Decryption 41
3.4 Advanced Encryption Standard (AES) 42
3.5 Format Preserving Encryption and Decryption (using AES 256) 45
3.6 Format and Type Preserving Encryption and Decryption for ALPHNUMERIC (using AES 256)47
3.7 GPG Encryption and Decryption 52
3.8 Triple Data Encryption Standard (3DES) 53

4 ASCII De-identification and Re-identification 55

5 Encoding and Decoding 55

6 Hashing 56

7 String Masking (Redaction) 57
7.1 Defining a Mask 58
7.1.1 Replace characters 58
7.1.2 Format Strings 59
7.2 Using a Mask 60
7.2.1 Universally Unique Identifier 61

8 Pseudonymization 62

9 SET Files 62
9.1 Creating SET Files 63
9.1.1 Character 63
9.1.2 Numeric 63
9.1.3 Date 64
9.1.4 Multi-column 64
9.2 Using SET Files in a Job Script 65

10 Tokenization 72

11 Randomization 72
11.1 Alpha-numeric list of values 72
11.2 Numeric values and ranges 72
11.3 Date values and ranges 73

12 /Re-formatting and Masking Options 74
12.1 Find and Replace 78
12.2 Named Fields With a Literal (fixed) Value 81

13 String-level Obfuscation 81
13.1 ASCII Substrings 81
13.1.1 INSTR 82

13.2 Date Intervals 83

14 Mathematical Data Obfuscation 84

15 Custom Masking Functions 87
15.1 Writing Libraries For Custom Data Protection 88
15.2 Syntax 88
15.3 Custom Procedure Declaration 89
15.4 Custom Procedure Arguments 89
15.4.1 Source Arguments 90
15.4.2 Result Argument 92
15.4.3 Return Value 93
15.5 Sequence of Custom Procedure Calls 94
15.5.1 Initialization Call 94
15.5.2 Processing field-level transformations for each record 94
15.5.3 Termination 95
15.5.4 Example 95

16 FieldShield Examples 101

17 Auditing 121

Dynamic Data Masking 129

1 Overview 129
1.1 FieldShield Library Process 129
1.2 FieldShield Library Main Classes 130

2 Java 131
2.1 Command line 131
2.2 Eclipse 131
2.3 FieldShield Library Usage 135

3 Microsoft Visual Studio 135
3.1 Installing the FieldShield Library 135

FieldShield Language Basics 139

1 Files 139
1.1 Data Files 139
1.1.1 Input Files 139
1.1.2 Output Files 142
1.2 Data Definition Files 143
1.3 Specification Files 144

2 Fields 145
2.1 Syntax: Input /FIELD Definitions 146
2.2 Syntax: Output /FIELD definitions 146
2.3 Field Name 147
2.4 POSITION 148
2.5 SEPARATOR 150
2.6 SIZE 153
2.7 PRECISION 154
2.8 FRAME 155
2.9 Alignment 157
2.10 Trimming 158
2.11 FILL 158
2.12 Data Types (Single-Byte) 159
2.13 Multi-Byte Data Types 159

3 /INREC 161

4 Conditions 162
4.1 Syntax 163
4.2 Binary Logical Expressions 163
4.3 Compound Logical Expressions 164
4.4 Perl-Compatible Regular Expressions 165
4.5 Condition Evaluation Order 166
4.6 Compound Conditions 166

5 Data Source and Target Formats (/PROCESS) 167
5.1 RECORD or RECORD_SEQUENTIAL 168
5.2 MFVL_SMALL 169
5.3 MFVL_LARGE 170
5.4 VARIABLE_SEQUENTIAL (or VS) 170
5.5 LINE_SEQUENTIAL (or LS) 170
5.6 VISION 171
5.7 VSAM 171
5.8 MFISAM 171
5.9 UNIVBF 171
5.10 CSV 172
5.11 LDIF 172
5.12 ODBC 173
5.12.1 Syntax: Using /PROCESS=ODBC 174
5.12.2 EXT_FIELD 175
5.12.3 Extraction 176
5.12.4 Loading 177

5.12.5 Special Considerations 179
5.13 BLOCKED 180
5.14 XML 180
5.15 Web Logs 181

6 /LOCALE 182

Custom Procedures 187

1 Custom Input 187
1.1 Syntax: /INPROCEDURE 187
1.2 Custom Input Procedure Declaration 187
1.3 Building the Custom Input Procedure for Runtime Linking 188

2 Custom Output 188
2.1 Syntax: /OUTPROCEDURE 189
2.2 Custom Output Procedure Declaration 189
2.3 Building the Custom Output Procedure for Runtime Linking 190

3 Example: Using Custom Input and Output Procedures 190

FieldShield Metadata Tools 193

1 cob2ddf 194
1.1 Usage 194
1.2 Example 195

2 csv2ddf 196
2.1 Usage 196
2.2 Example 197

3 ctl2ddf 198
3.1 Usage 198
3.2 Example 198

4 elf2ddf 200
4.1 Usage 200
4.2 Example 201

5 ldif2ddf 203
5.1 Usage 203
5.2 Example 204

6 xml2ddf 205
6.1 Usage 205
6.2 Example 206

7 odbc2ddf 207
7.1 Basic Syntax 207
7.2 Complete Syntax 207
7.3 Example 208

8 CLF Templates 211
8.1 Usage 211
8.2 Examples 211
8.2.1 Common (Access) Log 211
8.2.2 Referral Log 212
8.2.3 Agent Log 213

Upgrade Options 215

Appendix 217

A Data Types 217
A.1 Form 0 -- Alphabetic Data Types 217
A.2 Form 1 -- The Numeric Types 224
A.2.1 C Types 225
A.2.2 RM COBOL Data Types 229
A.2.3 MF COBOL Data Types 231
A.2.4 EBCDIC Native RM COBOL Data Types 233
A.2.5 EBCDIC Native MF COBOL Data Types 233
A.3 Forms 2-4 -- Date/Time/Timestamp Data Types 234

B Error Values 235

C ASCII Collating Sequence 239

D EBCDIC Printing Characters 240

FieldShield Index 241

Introduction to FieldShield

1 What is FieldShield?

FieldShield is a software platform for masking personally identifiable information (PII) for data at rest (static data masking) and data in motion (dynamic data masking). FieldShield discovers and protects PII and other sensitive data in databases and files in compliance with governmental data privacy regulations and business rules.

FieldShield's primary value is data-centricity. Users choose from a range of available functions, and apply them to individual data elements in database tables (columns) and files (fields). This aligns the ciphertext of each target field with its intended use and recipient authorization; e.g., pseudonymize names, encrypt credit card numbers, and partially redact the phone numbers. This same approach can also be used to secure an entire data source with a single protection function.

1.1 FieldShield Features

FieldShield enables you to do the following:

- Find PII in database, file, and dark (document) sources*
- Apply one or more reversible (or not) protection functions at once
- Apply these functions at the field, record, or file level
- Maintain the appearance of secure data with pseudonymization and format-preserving encryption
- Generate multiple, secure targets in one pass through one or more data sources
- Preserve referential integrity through consistent application of functions on like fields (rules) *
- Generate an XML audit log with each runtime to verify compliance
- Modify, re-use, and execute job with different methods, and on multiple operating systems
- Design, debug, deploy, manage, share, and schedule jobs *

* Requires the IRI Workbench GUI for FieldShield

The word "field" refers to a single data element, such as name, phone, zip, etc. In FieldShield, a field is any area of a record ranging from a single byte to the entire record. Fields can also be over-defined; e.g., field X covers datum A, B, C -- and field Y is B, C, D. Field X can have one encryption key, and Field Y can have another.

A principal benefit of FieldShield is that the protection of fields is selective. When encryption is the chosen protection method, the same algorithm can be used, but with different keys, or, different algorithms can be used. Encryption strength varies from clear (unencrypted) to GPG, to AES-256, and to user-defined algorithms. Other benefits are the speed in volume from I/O techniques developed for its underlying CoSort technology, and the simultaneous re-formatting features of the CoSort SortCL program syntax that FieldShield scripts inherit.

With FieldShield, the same target (file or table) can be sent to multiple recipients safely, because only those with the right decryption key can read their data. One pass over the data reduces production and transmission time, and eliminates data redundancy synchronization. FieldShield allows one field of characters to be replaced by another. For example, "Joe Smith" becomes "Sam Houston". This makes the mapped names look real. One-to-one mappings can be recovered.

1.2 FieldShield User Recommendations

FieldShield delivers more than 12 different categories of data masking functions. To apply them appropriately, you must know the data privacy laws and business rules operating on the organization, and sometimes how they apply to a larger information stewardship initiative or enterprise information (data lifecycle management) architecture. To apply them effectively, you must understand whether your requirements are static or dynamic, and be able to make use of the data discovery and job design features available to you. Thus administrators and data governance teams should consider:

- *Forming* an ongoing liaison between the generators and consumers of data to develop and maintain the data security requirements of the enterprise. They will catalog the above knowledge in a user <-> data matrix (spreadsheet) describing the locations of (discovered) data, and which users can see what. This is an ongoing effort because new data are introduced regularly and personnel are always changing. On a regular basis, the need-to-know recipients of protected data need to be apprised of the current methods (e.g., decryption job and key) required to reveal the original values.
- *Profiling* the data through string, pattern, and fuzzy-matching search and classification features available in free FieldShield GUI data discovery wizards. And, Acquiring or building the metadata that describe the physical characteristics (table, file, format, data type ...) of the data. The work is made easier when retrofitting security into existing applications. FieldShield has routines available that can preserve original data formats.
- *Selecting* one or more of the encryption algorithms available in FieldShield (or creating a custom algorithm) that supports reversibility requirements in the matrix. FieldShield programs will either read clear data (unencrypted plaintext) and produce protected data (e.g., ciphertext), or when the decryption key is provided to a subsequent job script, FieldShield will read the encrypted data (ciphertext), and produce plaintext.
- *Developing* FieldShield job control scripts. The syntax and semantics of FieldShield is straightforward; most users learn quickly after seeing a few examples. FieldShield protects columns within ODBC-connected tables and

popular flat-file formats (e.g., TXT, CSV, LDIF, DAT, SAM). Extensive file format and field data types are also recognized.

- *Generating* runtime statistics and audit logs so you can monitor performance and verify compliance with privacy regulations with a record of every job touching what data, in what way, where, when, and by whom.

2 FieldShield Technical Specifications

Functions

- multiple, NSA Suite B and FIPS-compliant encryption (and decryption) algorithms, including format-preserving encryption
- SHA-1 and SHA-2 hashing
- ASCII de-ID
- binary encoding
- randomization
- canned and custom data masking
- reversible and non-reversible pseudonymization
- custom expression (cross-calculation) logic
- filtering or redaction
- data type and file-format conversion
- byte shifting and substring functions
- tokenization (for PCI)

Inputs

Sources	any number of files, standard input (unnamed pipes), named pipes (in UNIX, Linux), ODBC-connected databases, and procedure calls. With partner drivers, and myriad legacy (proprietary application and mainframe), and modern (cloud, big data and SaaS) platforms listed here.
Volume	No row /size limit.
Record Length	1 ≤ fixed-length ≤ 65,535 bytes. 0 ≤ variable-length ≤ 65,535 bytes.
Program	Optional, user-written routines for reading special sources.
VLDB Bulk Unload	IRI FACT (Fast Extract) produces data definition file (DDF) metadata for the flat-files available for FieldShield operations
Data Types	The following categories of date types are supported:

Character strings	ASCII, EBCDIC, ASCII in EBCDIC order, days of the week, months of the year, etc.
Numerics	ASCII-Numeric, MF COBOL types, RM COBOL types, integers, floats, doubles, zoned decimal, etc.
Timestamps	American, European, ISO, Japanese, and current time, date, and timestamp.
Multi-byte	Double-byte types such as Shift JIS, EHANGUL, and Big 5.

Outputs

Terminal	stdout
Table	ODBC-connected database targets, and see the above source list.
File	User-selected name(s) or device(s), including input file overwrite, and create or append logic, plus named and unnamed pipes.
Both	Standard output and file to observe and abort unintended executions.
Program	Optional, user-written routines for writing to special targets.

Operating Optional Modes:

- standalone or embedded batch execution
- user program integration

Platforms

- AIX 5.1 and above on IBM Power architecture
- HP-UX 11.0 and above on the Intel Itanium architecture
- Linux kernel 2.6 and above on Intel x86 & x64 architecture
- Linux kernel 3.6 and above on ARMv6 and above architecture
- Solaris 5.8 and above on Sun Sparc architecture
- Solaris 10 on Intel x64 architecture
- Windows XP and above on Intel x86 & x64 architecture
- Windows Server 2003 and above on Intel x86 & x64 architecture

3 FieldShield Package Contents

The **FieldShield** package consists of the following utilities:

FieldShield Command-Line Interface (CLI)

*fieldshield** (Unix executable) or fieldshield.exe on Windows for Static Data Masking The FieldShield program reads FieldShield Control Language (FCL) job scripts based on the same JCL and SQL-oriented syntax of the CoSort Sort Control Language (SCL) program. FCL scripts -- and the IRI data definition file (DDF) metadata they reference -- specify simultaneous, multi-source, multi-target field-centric protection jobs, with one or more targets in one or more database or user-defined formats.

FieldShield Graphical User Interface (GUI)

IRI Workbench, for FieldShield program

The IRI Workbench GUI for FieldShield, is in the same Eclipse integrated development environment (IDE) for all IRI products, including the antecedent CoSort product, and the larger data management platform around both (IRI Voracity). The Workbench facilitates the discovery of PII through data profiling wizards, as well as the design, debugging, and deployment of FCL scripts through end-to-end job creation and metadata definition wizards, and a syntax-aware script editor with dynamic outlining. Function-specific dialogs can be used in lieu of scripting. Voracity users can also manage FCL jobs in mapping and workflow diagrams, automate them with an onboard task scheduler, and share them in local or cloud-based repositories like EGit.

FieldShield Software Development Kit (SDK)

C/C++, Java, and .NET API Libraries for Dynamic Data Masking

Application developers who need to apply encryption, string masking (redaction), or hashing functions to data in motion can pass the data into and out of these functions in memory in multiple languages. The SDK includes the dynamic load libraries and documentation necessary for establishing in-memory data flows that follow the movement, manipulation, and identity access rules of your application.

Metadata Conversion Tools

FieldShield CLI and GUI packages also include these executables to facilitate the use of existing metadata from certain data sources:

cob2ddf	Converts COBOL copybook layouts into a **FieldShield** data definition file (see the cob2ddf *chapter on page 194*).
csv2ddf	Examines headers of a Microsoft CSV (Comma-Separated Values) file and generates a **FieldShield** data definition file (see the csv2ddf *chapter on page 196*).
ctl2ddf	Converts Oracle SQL*Loader control file metadata into a **FieldShield** data definition file (see the ctl2ddf *chapter on page 198*).

elf2ddf	Examines headers of a web transaction file in *Extended Log Format* and generates a **FieldShield** data definition file (see the elf2ddf *chapter on page 200*).
odbc2ddf	Converts database table layouts into a **FieldShield** data definition file (see the odbc2ddf *chapter on page 207*), provided the database is compatible with ODBC (Open Database Connectivity).
xml2ddf	Converts XML data layouts into a **FieldShield** data definition file (see the xml2ddf *chapter on page 205*).

4 FieldShield Control Language (FCL) Purpose

The following describes the FieldShield Control Language (FCL) used by the FieldShield program. FCL syntax uses familiar keywords and logical expressions to define and transform large volumes of data, while offering several options for field-level protection.

NOTE

If you are new to FieldShield, it is recommended that you first familiarize yourself with FCL scripting through the sample programs in the final section of this chapter (see FieldShield Examples *on page 101*). This will be the fastest way to learn how to create your own job scripts without using wizards in the GUI. The other sections of this chapter are useful as a complete reference guide to the numerous features available within the FieldShield program and the FCL.

A single pass through the program can perform multiple field protection (data masking) operations. An input source can also undergo format conversion, and/or remap into one or more output targets. Any number of output layouts can also be defined.

5 Execution

fieldshield (fieldshield.exe on Windows) is a standalone program that is executed from the command line, from within a batch script, or from IRI Workbench 'Run' menus.

To begin execution from the command line, enter the program name fieldshield followed by a forward slash (/) and the name of a job specification file.

The syntax for executing a script from the command line is:

```
        fieldshield /SPECIFICATION=[path]script_file
```

`/SPECIFICATION` can be abbreviated as in the following example:

```
        fieldshield /spec=/home/test/job10.fcl
```

To display the version and other information, enter:

```
        fieldshield /v
```

This will display something like:

```
FieldShield v4.x.x D95390618-1454 64B Copyright 2009-2016
IRI, Inc. www.iri.com
EDT 12:19:18 Mon Feb 15 2016. # 0 CPU x86_64 Monitor Level 0
```

W The syntax required for Windows users referencing the path to a script_file depends on whether the drive letter is used. The following is an example of the syntax required in each case:

```
        fieldshield /specification=C:\\home\\test\\job10.fcl
        fieldshield /specification=/home/test/job10.fcl
```

Windows users must use a double backslash when including the drive letter as part of a /SPECIFICATION= command-line statement. If the drive letter is not used, you can use either a single forward slash (for consistency with UNIX) or a single backslash (the standard Windows convention). See Spaces within File Names/Paths -- Windows Users Only on page 24 if the name of the job script, or its path, contain a space. ◆

During **FieldShield** execution, any syntax errors are reported by the line number in your script.

6 Conventions

6.1 Documentation Conventions

This section describes conventions that are found throughout this user manual and other manuals provided by Innovative Routines International (IRI), Inc.

6.1.1 UNIX / Linux or Windows comments

Special comments that pertain only to UNIX and Linux users are preceded with the symbol **U** and are terminated with the symbol ◆.

Special comments that pertain only to Windows users are preceded with the symbol **W** and are terminated with the symbol ◆.

6.1.2 Italics: Documentation Convention

A word that is italicized throughout this documentation represents a variable that is to be replaced by a literal string, for example:

/INFILE=*filename*

indicates that the user might have, for example:

```
/INFILE=Inventory.dat
```

6.2 FCL Syntax Conventions

This section describes notation that is used in documenting FCL job scripts.

6.2.1 Naming Conventions

The following rules apply both to names and statements recognized by FieldShield, and to file names and field names that you define:

- must start with an alphanumeric character
- may be any length
- may contain any combination of the following characters:
  - alphabetic
  - numeric
  - embedded underscore (_)

Statements and field names are not case-sensitive, that is, upper- and lower-case letters are interchangeable, for example:

- `POSITION` is the same as `position`
- `/FIELD=(PARTY)` is the same as `/field=(Party)`

For Windows users with Fast Access Table (FAT) and NT File Systems (NTFS), file names are not case-sensitive, so the file **chicago** *is* the same as the file **CHICAGO**, **Chicago**, etc.

Refer to your operating system manual for the acceptable maximum length and naming format of a file name.

UNIX paths and file names, however, are case-sensitive, so the file **chicago** is *not* the same as the file **CHICAGO**, **Chicago**, etc.

The file names **stdin** and **stdout** are used for *standard input* and *standard output* (pipes), respectively. When input and output files are not specified, the defaults are **stdin** and **stdout**.

6.2.2 Abbreviations

Generally, **FieldShield** words in statements can be truncated by deleting trailing letters.
The control word must only be long enough to distinguish it from any other control word, for example:

`/FIELD=(Goods,POSITION=10)` is the same as
`/FIELD=(Goods,POS=10)`

and:

`/SPECIFICATIONS` is the same as
`/SPEC`

6.2.3 Optional Statement Parameters

Square brackets `[]` are used to describe optional parameters or values.

6.2.4 Environment Variables

The character `$` preceding a variable name directs **FieldShield** to replace the environment variable with its current value. You may use any of the following conventions:

- `$`*`variable`*
- `${`*`variable`*`}`
- `$[`*`variable`*`]`

UNIX users defining environment variables in a Bourne or Korn shell must export the variables for **FieldShield** to recognize them.

6.2.5 Line Continuation

The backslash character \ is the continuation character for FCL specification statements longer than one line. In a FieldShield script, you can include blanks, comments (starting with #), or tabs on the same line following the \ character. However, the first character on the next line must begin with the continuation of the syntax that was interrupted by the \. On the command line, UNIX users can use a backslash as a line continuation character, but spaces, tabs and comments are not supported beyond the \.

Windows users cannot use \ on the command line. Long lines should wrap around to the next line.

NOTE

You cannot use a \ line continuation character to separate a `[`*`path`*`]` *`file-name`* reference, even if it contains spaces (see Spaces within File Names/Paths -- Windows Users Only *on page 140*) Also, you cannot use the line continuation character to break up any word. You must place the \ before or after the complete word, and complete the statement on the next line.

6.2.6 Comments

The character `#` marks the beginning of comments on a line in a **FieldShield** script. Comments may begin after a statement on the same line, or may be on a line by themselves. The comment continues until the end of the line, and is ignored during processing.

Static Data Masking Functions

Static data masking is the primary method of protecting specific data elements at rest. These elements are typically database column or flat-file field values that are considered sensitive. These fields may contain personally identifiable information (PII), protected health information (PHI), or other sensitive, secret data.

1 Conditional Field Security and Masking

FieldShield permits the value for `/FIELD` statements in output file descriptions, and the value for `/DATA` statements in output file descriptions to be derived from `IF-THEN-ELSE` logic. When using this logic, the syntax for these statements is:

```
/FIELD=(fieldname[,field attributes][,IF-THEN-ELSE])
or /DATA=IF-THEN-ELSE
```

The general form for `IF-THEN-ELSE` is:

```
IF condition THEN value1 ELSE value2
```

If the condition is true, the resultant value of the `/FIELD` or `/DATA` statement is `value1`; if the condition is false, the resultant value is `value2`.

A `condition` is a named condition or a logical expression as discussed in *Conditions starting on page 162*. A condition can also use Perl-compatible regular expressions (see Perl-Compatible Regular Expressions *on page 165*).

A `value` can be any of the following:

- a field name
- a literal (numeric value or character string)
- a repeated string such as {180}"0" to indicate 180 zeroes (see /Re-formatting and Masking Options *on page 74*)
- an algebraic expression
- another `IF-THEN-ELSE` derived value (see Multiple IF THEN clauses *on page 20*).

The following is an example with a `/DATA` and a `/FIELD` statement:

```
/CONDITION=(Age,TEST=(type == "senior"))
/FIELD=(client, POSITION=2, SIZE=4, IF Age THEN member ELSE "standard")
/DATA=IF Age THEN instructions ELSE "none"
```

where `Age`, `client`, and `member` are previously defined input fields, and `"senior"` and `"standard"` are string literals. Notice that the string must be in double quotes.

Also, `value2` can be a recurring constant string such as `{5}"-"` -- that will display five dashes. Recurring constant strings are also accepted within `/DATA` statements (see /Re-formatting and Masking Options *on page 74*).

You can obtain the same results as above using implicit conditions:

```
/FIELD=(client, POSITION=2, SIZE=4, IF type == "senior" THEN member ELSE "standard")
/DATA=IF type == "senior" THEN instructions ELSE "none"
```

There are two variations on the general form of `IF-THEN-ELSE` logic:

- `IF` *`condition`* `THEN` *`value1`*
- `IF` *`condition`* `ELSE` *`value2`*

The first implies an empty value for the `ELSE` clause, and the second implies an empty value for the `THEN` clause. Below are examples:

```
/FIELD=(client, POSITION=2, SIZE=4, IF type == "senior" THEN member)
/DATA=IF type == "senior" ELSE "none"
```

These are equivalent to:

```
/FIELD=(client, POSITION=2, SIZE=4, IF type == "senior" THEN member ELSE "")
/DATA=IF type == "senior" THEN "" ELSE "none"
```

1.1 Multiple IF THEN clauses

A next level of `IF-THEN-ELSE` may appear in either or both of the `ELSE` or `THEN` clauses. Any number of levels can be defined to meet the degree complexity of a problem.

For example, the following is valid syntax:

```
IF C1 THEN IF C2 THEN V1 ELSE V2 ELSE IF C3 THEN V3 ELSE V4
```

but for clarity should be written:

```
IF C1 \
THEN IF C2 \
     THEN V1 \
     ELSE V2 \
ELSE IF C3 \
     THEN v3 \
     ELSE V4
```

The rule is that each `THEN` or `ELSE` clause is associated with the most recent `IF` that does not already have a `THEN` or `ELSE` clause associated with it. The line continuation at the end each line of the statement is necessary for **FieldShield** to evaluate the statement correctly.

NOTE

When long lists are to be tested for a true condition, processing will be faster if the cases that are most likely to be true are tested first.

In cases where a job script requires multiple THEN or ELSE clauses to return literal strings (that is, known values), it is recommended that you instead use **FieldShield**'s SET column look-up capability to perform key-value substitutions, as described in Universally Unique Identifier *on page 61*.

1.2 Conditional Field and Data Statement Examples

This section provides examples of using conditional field and conditional data statements. The following input file, **seqdata.in**, is used in these examples:

```
01 775-17-0363 Stuart Clay       56681.42 6 CT 101 B St
02 810-90-1269 Taylor Guerrero   15019.10 9 MD 1031 Park Ln Apt D
03 906-43-3545 Charles Caldwell  41116.71 3 NY 14 Main St
04 787-73-7773 Robyn Puckett     44558.62 3 NY 822 Hwy 76
05 950-52-1240 Santiago Lindsey  55836.11 0 TX Star Rt Box 822
06 891-56-9799 Charles Lindsey   98525.58 2 TX 12746 Wolf Circle
07 789-12-7387 Santiago Puckett  59036.80 4 NY 321 Baltic Ct
08 899-60-4065 Charles Williams  18645.95 1 TX 1103 Fresh Creek Ln
09 951-05-4521 Jack Velazquez    29205.44 2 NY 6780 Sand Dr Apt 3A
10 788-35-4977 Donald Cooke      44121.44 4 MA 35 La Palma Dr
```

and the following metadata (data definition file) is used, **seqdata.ddf**:

```
/FIELD=(idnum,POSITION=1,SIZE=2.0,NUMERIC)
/FIELD=(ssno,POSITION=4,SIZE=11)
/FIELD=(name,POSITION=16,SIZE=18)
/FIELD=(salary,POSITION=34,SIZE=8,NUMERIC)
/FIELD=(IDno,POSITION=43,SIZE=1)
/FIELD=(state,POSITION=45,SIZE=2)
/FIELD=(address,POSITION=48,SIZE=20)
```

Example 1 Simple If-Then-Else Condition

The following script, **simple_data.fcl**, is a simple case of using an If-Then-Else condition statement to identify indiviuals from New York:

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf
/REPORT
/OUTFILE=newyorkers.out
    /FIELD=(name, POSITION=1, SIZE=18)
    /FIELD=(state, POSITION=20, SIZE=2)
   /FIELD=(snark, POSITION=23, IF state == "NY" THEN "New Yorker"\
ELSE "Other State")
```

The resulting output is:

```
Stuart Clay        CT  Other State
Taylor Guerrero    MD  Other State
Charles Caldwell   NY  New Yorker
Robyn Puckett      NY  New Yorker
Santiago Lindsey   TX  Other State
Charles Lindsey    TX  Other State
Santiago Puckett   NY  New Yorker
Charles Williams   TX  Other State
Jack Velazquez     NY  New Yorker
Donald Cooke       MA  Other State
```

Example 2 Multi-Level Condition Based on One Field

This job script, **multi_one.fcl**, uses a multi-level condition for an output field where the test is based on the value of one input field. This is the equivalent of `ELSIF` statements in some other languages.

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf
/REPORT
/OUTFILE=state_ids.out
    /FIELD=(name, POSITION=1, SIZE=18)
    /FIELD=(state, POSITION=20, SIZE=2)
    /FIELD=(snark, POSITION= 23, \
    IF state == "NY" \
        THEN "New Yorker" \
    ELSE IF state == "TX" \
        THEN "Texan" \
    ELSE IF state == "MD" \
        THEN "Marylander" \
    ELSE "Other")
```

The output is:

```
Stuart Clay         CT Other
Taylor Guerrero     MD Marylander
Charles Caldwell    NY New Yorker
Robyn Puckett       NY New Yorker
Santiago Lindsey    TX Texan
Charles Lindsey     TX Texan
Santiago Puckett    NY New Yorker
Charles Williams    TX Texan
Jack Velazquez      NY New Yorker
Donald Cooke        MA Other
```

Example 3 Multi-Level Condition Based on Two Fields

This job script, **multi_two.fcl**, uses a multi-level condition for an output field where the test is based on the value of two input fields in some instances (salary and state), and on one input field (state) in other instances. Notice that both `THEN IF` and `ELSE IF` are used.

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf
/REPORT
/OUTFILE=multi_cond.out
    /FIELD=(name, POSITION=1, SIZE=18)
    /FIELD=(state, POSITION=20, SIZE=2)
    /FIELD=(salary, POSITION=23, SIZE=8, PRECISION=2, NUMERIC)
    /FIELD=(snark, POSITION= 32, \
      IF state == "NY" \
      THEN IF salary > 50000 \
        THEN "High-paid New Yorker" \
        ELSE "Low-paid New Yorker" \
    ELSE IF state == "TX" \
        THEN IF salary > 50000 \
            THEN "High-paid Texan" \
            ELSE "Low-paid Texan"\
        ELSE IF state == "MD" \
            THEN "Marylander" \
            ELSE "Other")
```

The output is as follows:

```
Stuart Clay         CT 56681.42  Other
Taylor Guerrero     MD 15019.10  Marylander
Charles Caldwell    NY 41116.71  Low-paid New Yorker
Robyn Puckett       NY 44558.62  Low-paid New Yorker
Santiago Lindsey    TX 55836.11  High-paid Texan
Charles Lindsey     TX 98525.58  High-paid Texan
Santiago Puckett    NY 59036.80  High-paid New Yorker
Charles Williams    TX 18645.95  Low-paid Texan
Jack Velazquez      NY 29205.44  Low-paid New Yorker
Donald Cooke        MA 44121.44  Other
```

2 Record Filtering (Omission)

`/INCLUDE` and `/OMIT` statements use conditions to accept or reject entire records, respectively. This feature can be used to protect entire details associated with a specifically identifiable person or group. Include and omit conditions use field-value conditions to determine the dispositions of entire records. This functionality can be applied to records for both input filtering and/or output purposes.

2.1 Syntax

The forms of `/INCLUDE` and `/OMIT` statements are:

```
/INCLUDE WHERE condition
/INCLUDE=(CONDITION=condition)
/OMIT WHERE condition
/OMIT=(CONDITION=condition)
```

where *`condition`* is a logical expression or a condition name, as discussed in *Conditions starting on page 162*.

2.2 Include-Omit Statement Order Evaluation

Care must be given to the order of `/INCLUDE` and `/OMIT` statements because records are tested for each `/INCLUDE` and `/OMIT` condition in the order specified. If a particular record meets a given `/INCLUDE` condition, it is included, *regardless* of any remaining
`/OMIT` statements that would otherwise cause that record to be omitted. Alternatively, if a record meets a given `/OMIT` condition, it is omitted, regardless of any remaining
`/INCLUDE` statements that would otherwise cause that record to be included. That is, once a record satisfies an include or omit condition, its inclusion/exclusion in the output is determined.

For better performance when there are multiple conditions, the most likely `/INCLUDE` and `/OMIT` statements should be given early. The sooner the records' dispositions are determined, the fewer conditions are required to be evaluated.

You should also consider whether the `/INFILE` or `/OUTFILE` section(s) of the script is better for placement of `/INCLUDE` and `/OMIT` statements.

Any records that are not required in the output should be filtered out prior to being processed to the output. This makes the job more efficient by keeping unnecessary records from being processed further. Be sure the record will not be required for deriving fields in the output.

2.3 Include-Omit Examples

This section provides examples of includes and omits.

The following input file, **seqdata_phone.in**, is used in these examples:

```
775-17-0363 Stuart Clay        (155)321-2345 CT
810-90-1269 Taylor Guerrero    (132)456-7123 MD
906-43-3545 Charles Caldwell   (191)812-7345 NY
787-73-7773 Robyn Puckett      (178)378-9567 NY
950-52-1240 Santiago Lindsey   (133)262-8096 TX
891-56-9799 Charles Lindsey    (127)734-1723 NY
899-60-4065 Charles Williams   (177)735-7458 TX
951-05-4521 Jack Velazquez     (127)834-6275 NY
788-35-4977 Donald Cooke       (177)683-1634 MA
```

Example 4 Using /INCLUDE with WHERE

This script, **include.fcl**, shows an `/INCLUDE` with an explicit condition using a WHERE clause:

```
/INFILE=seqdata_phone.in
    /FIELD=(ssno, POSITION=1, SIZE=11)
    /FIELD=(name, POSITION=13, SIZE=18)
    /FIELD=(areacode, POSITION=32, SIZE=3)
    /FIELD=(local_pt1, POSITION=37, SIZE=3)
    /FIELD=(local_pt2, POSITION=41, SIZE=4)
    /FIELD=(state, POSITION=46, SIZE=2)
    /INCLUDE WHERE state EQ "NY"
/REPORT
/OUTFILE=NY_only.out
```

The output is:

```
906-43-3545 Charles Caldwell   (191)812-7345 NY
787-73-7773 Robyn Puckett      (178)378-9567 NY
891-56-9799 Charles Lindsey    (127)734-1723 NY
951-05-4521 Jack Velazquez     (127)834-6275 NY
```

Example 5 Using /INCLUDE with Named Conditions

The following script, **include_names.fcl**, shows two /INCLUDE statements with named conditions:

```
/INFILE=seqdata_phone.in
    /FIELD=(ssno, POSITION=1, SIZE=11)
    /FIELD=(name, POSITION=13, SIZE=18)
    /FIELD=(areacode, POSITION=33, SIZE=3)
    /FIELD=(local_pt1, POSITION=37, SIZE=3)
    /FIELD=(local_pt2, POSITION=41, SIZE=4)
    /FIELD=(state, POSITION=46, SIZE=2)
    /CONDITION=(newyorkers, TEST=(state EQ "NY"))
    /CONDITION=(area177, TEST=(areacode EQ "177"))
    /INCLUDE WHERE newyorkers
    /INCLUDE WHERE area177
/REPORT
/OUTFILE=ny177.out
```

and produces the following output:

```
906-43-3545 Charles Caldwell   (191)812-7345 NY
787-73-7773 Robyn Puckett      (178)378-9567 NY
891-56-9799 Charles Lindsey    (127)734-1723 NY
899-60-4065 Charles Williams   (177)735-7458 TX
951-05-4521 Jack Velazquez     (127)834-6275 NY
788-35-4977 Donald Cooke       (177)683-1634 MA
```

In this case, records are included that satisfy either the `newyorkers` condition or the `area177` condition. The rest are excluded.

Example 6 Using Two /OMIT Statements

This script, **omits.fcl**, uses two omits. One is used with a named condition in a `WHERE` clause; the other is used with an explicit logical expression in a `WHERE` clause.

```
/INFILE=seqdata_phone.in
    /FIELD=(ssno, POSITION=1, SIZE=11)
    /FIELD=(name, POSITION=13, SIZE=18)
    /FIELD=(areacode, POSITION=33, SIZE=3)
    /FIELD=(local_pt1, POSITION=37, SIZE=3)
    /FIELD=(local_pt2, POSITION=41, SIZE=4)
    /FIELD=(state, POSITION=46, SIZE=2)
    /CONDITION=(newyorkers, TEST=(state EQ "NY"))
    /OMIT WHERE newyorkers
    /OMIT WHERE areacode EQ "177"
/REPORT
/OUTFILE=nony_no177.out
```

The output is:

```
775-17-0363 Stuart Clay        (155)321-2345 CT
810-90-1269 Taylor Guerrero    (132)456-7123 MD
950-52-1240 Santiago Lindsey   (133)262-8096 TX
```

In this case, records that satisfy either the `newyorkers` condition or the `areacode EQ "177"` condition are omitted. The remaining three records are included.

As the previous two examples illustrate, in cases where there are only `/INCLUDE` statements and no `/OMIT` statements, all records that do not satisfy the `/INCLUDE` conditions are discarded. Alternatively, if you have only `/OMIT` statements in your script, all records that do not satisfy the `/OMIT` conditions are included.

When mixing two or more `/INCLUDE` and `/OMIT` statements, there is the possibility that a record will not satisfy any test. If this happens, the disposition of the record depends on the last test. If the final test was an `/INCLUDE`, the record is omitted. If the final test was an `/OMIT`, the record is included.

To illustrate this concept, the following two examples use the same conditions but in different order. The `OverTen` condition is true when `Price` > 10.00. The condition `YesDell` is satisfied when a book is published by Dell. The following two examples yield different results.

Example 7 Condition Ordering: /INCLUDE first

The following script, **include_first.fcl**, will first test the record to see if the condition `state EQ "NY"` is true. If it is, the record will be included in the output, regardless of subsequent include or omit statements. If the condition `state EQ "NY"` is not true, the record is tested for the condition `areacode EQ "177"`. If this test is true, that record is omitted. Otherwise, it will be included.

```
/INFILE=seqdata_phone.in
    /FIELD=(ssno, POSITION=1, SIZE=11)
    /FIELD=(name, POSITION=13, SIZE=18)
    /FIELD=(areacode, POSITION=33, SIZE=3)
    /FIELD=(local_pt1, POSITION=37, SIZE=3)
    /FIELD=(local_pt2, POSITION=41, SIZE=4)
    /FIELD=(state, POSITION=46, SIZE=2)
    /INCLUDE WHERE state EQ "NY"
    /OMIT WHERE areacode EQ "177"
/REPORT
/OUTFILE=ny_no177
```

The output is:

```
775-17-0363 Stuart Clay        (155)321-2345 CT
810-90-1269 Taylor Guerrero    (132)456-7123 MD
906-43-3545 Charles Caldwell   (191)812-7345 NY
787-73-7773 Robyn Puckett      (178)378-9567 NY
950-52-1240 Santiago Lindsey   (133)262-8096 TX
891-56-9799 Charles Lindsey    (127)734-1723 NY
951-05-4521 Jack Velazquez     (127)834-6275 NY
```

Example 8 Condition Ordering: /OMIT first

In the following script, **omit_first.fcl**, a record is first tested for the condition `areacode EQ "177"`, and if true, the record is omitted. If `areacode EQ "177"` is not true, the next test is applied. If the record passes the condition `state EQ "NY"`, it is included in the output. But if it fails the `state EQ "NY"` test, the record is omitted because it failed all of the tests and the last test was an `/INCLUDE`.

```
/INFILE=seqdata_phone.in
    /FIELD=(ssno,POSITION=1,SIZE=11)
    /FIELD=(name,POSITION=13,SIZE=18)
    /FIELD=(areacode,POSITION=32,SIZE=3)
    /FIELD=(local_pt1,POSITION=37,SIZE=3)
    /FIELD=(local_pt2,POSITION=41,SIZE=4)
    /FIELD=(state,POSITION=46,SIZE=2)
    /OMIT WHERE areacode EQ "177"
    /INCLUDE WHERE state EQ "NY"
/REPORT
/OUTFILE=no177_ny.out
```

The output is different for this job because the order of the `/INCLUDE` and `/OMIT` statements is reversed.

The output is:

```
906-43-3545 Charles Caldwell   (191)812-7345 NY
787-73-7773 Robyn Puckett      (178)378-9567 NY
891-56-9799 Charles Lindsey    (127)734-1723 NY
951-05-4521 Jack Velazquez     (127)834-6275 NY
```

2.4 Record Filtering

/HEADSKIP

This statement causes **FieldShield** to skip the first *n* bytes of an input file before applying the file format used to define the records. The syntax is:

```
/HEADSKIP=n
```

For example, if **bookstock** begins with the header record:

```
Title          Publisher     Qty     Price
```

it can be skipped over before processing begins by using:

```
/INFILE=bookstock
   /HEADSKIP=43         # length of header
/REPORT
/OUTFILE=noTop
```

The output of **noTop** contains the file **bookstock**, minus the header.

/TAILSKIP

This statement causes **FieldShield** to skip the last *n* bytes of an input file before applying the file format used to define the records. The syntax is:

```
/TAILSKIP=n
```

For example, if **bookstock** ends with the record:

```
End of File
```

it can be skipped over before processing begins by using:

```
/INFILE=bookstock
   /TAILSKIP=12         # length of tail record
/REPORT
/OUTFILE=noBottom
```

The output of **noBottom** contains the file **bookstock**, minus the final record.

/INSKIP

This statement causes **FieldShield** to skip the first *n* number of records from the input file after any `/INCLUDE` or `/OMIT` conditions have been evaluated.

The syntax for `/INSKIP` is:

```
/INSKIP=n
```

where *n* is the number of remaining records to be skipped.

Example 9 Using /INSKIP

For example, using **bookstock** as the input (see *Example 11* on page 36), the following script, **inskip.fcl**:

```
/INFILE=bookstock
    /FIELD=(title, POSITION=1, SIZE=15)
    /FIELD=(publisher, POSITION=16, SIZE=13)
    /FIELD=(quantity, POSITION=30, SIZE=3, NUMERIC)
    /FIELD=(price, POSITION=38, SIZE=5, NUMERIC)
    /INCLUDE WHERE price GT 10
    /INSKIP=2
/REPORT
/OUTFILE=inskip.out
```

produces this output:

```
Sending Your   Valley Kill   130     15.75
People Please  Valley Kill    75     11.50
Map Reader     Prentice-Hall 200     14.95
```

In this case, the records with the titles `Reasoning For` and `Still There` were skipped because they were the first two input records that satisfied the condition.

/OUTSKIP

This statement causes FieldShield to skip the first *n* number of processed records after any `/INCLUDE` or `/OMIT` conditions have been evaluated. The syntax is:

```
/OUTSKIP=n
```

where *n* is the number of remaining records to be skipped.

/INCOLLECT

This statement determines the maximum number of records to be accepted and processed from an input source. This statement is placed in the script after any other input filters for the input file are specified. The syntax is:

```
/INCOLLECT=n
```

where *n* is the number of records to be processed after all `/INCLUDE`, `/OMIT`, and `/INSKIP` filters are applied.

/JOBSKIP

This statement causes **FieldShield** to skip the first *n* number of records from your input data prior to processing. It is typically used when you have multiple input sources.

`/JOBSKIP` operates on records that are still remaining after any `/INCLUDE` or `/OMIT` logic is satisfied, and after any `/INSKIP` and `/INCOLLECT` filters have been applied to individual inputs.

The syntax for `/JOBSKIP` is:

```
/JOBSKIP=n
```

where *n* is the number of remaining records to be skipped.

The order that you specify your input files within your script is important because a /JOBSKIP statement is applied to your input data in a sequential fashion. For example, if you have 4 input files consisting of 25 records each, and /JOBSKIP is set to 50, then only records from the third and fourth input files will be processed, provided that no other filter logic has excluded them. no other filter logic has excluded them.

/JOBCOLLECT

This statement determines the maximum number of records accepted from your input data. It is typically used to limit total processing volume when there are multiple input sources. `/JOBCOLLECT` operates on records that are still remaining after any `/INCLUDE` or `/OMIT` logic is satisfied, and, if specified, after a `/JOBSKIP` statement has been applied.

The syntax is:

```
/JOBCOLLECT=n
```

where *n* is the number of records to be processed.

The order in which you specify your input sources within your script is important because a /JOBCOLLECT statement is applied to your input data in a sequential fashion. For example, if you have 4 input files consisting of 25 records each, and /JOBCOLLECT is set to 50 (with no /JOBSKIP statement), then only records from the first and second inputs will be processed, provided that no other filter logic has excluded them.

/OUTCOLLECT

This statement sets the number of records sent to an output file after the `/INCLUDE`, `/OMIT`, and `/OUTSKIP` filters are applied. The syntax is:

```
/OUTCOLLECT=n
```

where *n* is the maximum number of records sent to the output file.

/CREATE

This is the default specification for an output file. It indicates that a new output file will be created. When used with an existing table, that table will be truncated. If the file name already exists, all previous data in the file will be lost, even if nothing is written by this job.

The syntax is:

```
OUTFILE=output_filename
   /CREATE
```

/APPEND

Associate an `/APPEND` with any target to cause output data to be placed after the existing data in a file or table. If the file does not exist or is empty, `/CREATE` will be invoked. This is the default specification for outputting to a table.

The syntax is:

```
/OUTFILE=output_filename
   /APPEND
```

2.5 Record Formatting

/HEADREC

This statement creates a new customized header record in the output file (report).

The syntax is:

```
/HEADREC="character string with format for each embedded \
  variable (format control characters) ..."[, var1, var2, ...]
```

The *`character string`* can be a constant that may contain any combination of internal variables and control (escape) characters recognized by **FieldShield** (see *Table 4* on page 76 and *Table 3* on page 75).

Some examples of using the `/HEADREC` statement are:

```
/HEADREC="The Monthly Report\n"
/HEADREC="%s               Sales Report\n",CURRENT_DATE
```

where `\n` indicates a new line, and `%s` is the format for the variable `CURRENT_DATE`. *See Table 4* on page 76 for a list of accepted variables.

NOTE The `\n` is needed to cause a line-feed. Without it, the first record will display immediately after the header on the same line. Also, the variables must be listed in the order that they will appear in the string.

Table 1 lists the display formats known to **FieldShield**.

Table 1: Format Control Characters for Variables

Character	Printed Result
`%c`	character
`%d, %i`	decimal integer
`%u`	unsigned decimal integer
`%o`	unsigned octal integer
`%x, %X`	unsigned hexadecimal integer
`%e`	floating point number, such as 4.321e+00
`%E`	floating point number, such as 4.321E+00
`%f`	floating point number, such as 4.321
`%g`	either e-format or f-format, whichever is shorter
`%G`	either E-format or f-format, whichever is shorter
`%s`	as a string
`%%`	writes a single `%` to the output stream; no argument is converted

/FOOTREC

This statement uses the same syntax as `/HEADREC` but its string values appear at the bottom of the output data as a footer (see /HEADREC *on page 32*).

The syntax is:

`/FOOTREC="`*`character string`* *with format for each embedded variable*`..."[, `*`var1, var2,`*` ...]`

The *`character string`* can be a constant that may contain any combination of internal variables, control (escape) characters, and conversion-specifier characters recognized by **FieldShield** (see *Table 4* on page 76 and *Table 3* on page 75).

NOTE

A constant string can also contain syntax specific to a mark-up language, such as HTML, provided the browser (or other utility) you use to read the output file accepts that syntax.

An example of using the `/FOOTREC` statement is:

```
/FOOTREC="generated by  %s [%d] -------\n\f",\ USER,PAGE_NUMBER
```

/HEADREAD

This statement reads and saves, but does not process, the first *n* bytes of an input file. These bytes will be used in the corresponding `/HEADWRITE` statement for an output file (see /HEADWRITE *on page 34*).

`/HEADREAD` is useful for output reporting where a special header record or formatted title needs to appear in the output report, but should not be processed with the file's records.

The syntax for `/HEADREAD` is:

```
/HEADREAD=n
```

where *n* is the number of bytes held for output.

/HEADWRITE

This statement causes a header record (the first record) that was scanned from an input file using `/HEADREAD` to be written in an output file. The header record is an exact copy of the header from a corresponding `/HEADREAD` statement associated with an input file (see /HEADREAD *on page 34*).

The syntax is:

```
/HEADWRITE=input filename
```

The `/HEADREAD` and `/HEADWRITE` statements can be used to transfer binary data as well as ASCII text.

NOTE

If the `/HEADREC` statement is used in addition to `/HEADWRITE`, the `/HEADREC` string is displayed second.

Example 10 Using /HEADREAD and /HEADWRITE

For this example, two 44-byte header lines were added to **bookstockhd**:

```
Title          Publisher     Qty     Price

Reasoning For  Prentice-Hall 150     10.25
Murder Plots   Harper-Row    160      5.90
Still There    Dell           80     13.05
Pressure Cook  Harper-Row    228      9.95
Sending Your   Valley Kill   130     15.75
People Please  Valley Kill    75     11.50
Map Reader     Prentice-Hall 200     14.95
```

To process the **bookstockhd** file while preserving header data, the specification file **hi-lo.fcl** contains:

```
/INFILE=bookstockhd
    /HEADREAD=88         # length of header data
    /FIELD=(title,POSITION=1,SIZE=15)
    /FIELD=(publisher,POSITION=16,SIZE=13)
    /FIELD=(quantity,POSITION=30,SIZE=3,TYPE=NUMERIC)
    /FIELD=(price,POSITION=38,SIZE=5,TYPE=NUMERIC)
/REPORT
/OUTFILE=hiprice.out
    /INCLUDE WHERE price GT 10
    /HEADWRITE=bookstockhd # header data
/OUTFILE=loprice.out
    /OMIT where price GT 10
    /HEADWRITE=bookstockhd # header data
```

The resulting output of **hiprice.out** is:

```
Title          Publisher     Qty     Price

Reasoning For  Prentice-Hall 150     10.25
Still There    Dell           80     13.05
Sending Your   Valley Kill   130     15.75
People Please  Valley Kill    75     11.50
Map Reader     Prentice-Hall 200     14.95
```

and the output of **loprice.out** is:

```
Title          Publisher     Qty     Price

Murder Plots   Harper-Row    160      5.90
Pressure Cook  Harper-Row    228      9.95
```

/TAILREAD

This statement reads and saves, but does not process, the last *n* bytes of an input file. These bytes will be used in the corresponding `/TAILWRITE` statement for an output file (see /TAILWRITE *on page 36*).

`/TAILREAD` is useful for output reporting where a special tail record needs to appear in the output report, but should not be processed with the file's records.

The syntax for `/TAILREAD` is:

```
/TAILREAD=n
```

where `n` is the number of bytes held for output.

/TAILWRITE

This statement causes a tail record (the last record) that was scanned from an input file using `/TAILREAD` to be written in an output file. The tail record will be an exact copy of the tail record from the corresponding `/TAILREAD` statement associated with an input file (see /TAILREAD *on page 36*).

The syntax is:

```
/TAILWRITE=input filename
```

The `/TAILREAD` and `/TAILWRITE` statements can be used to transfer binary data as well as ASCII text.

NOTE

If the `/FOOTREC` statement is used in addition to `/TAILWRITE`, the `/FOOTREC` string is displayed second.

Example 11 Using /TAILREAD and /TAILWRITE

For this example, the input file **bookstocktl** includes two 44-byte non-data records at the end of the file:

```
Reasoning For  Prentice-Hall 150     10.25
Murder Plots   Harper-Row    160      5.90
Still There    Dell           80     13.05
Pressure Cook  Harper-Row    228      9.95
Sending Your   Valley Kill   130     15.75
People Please  Valley Kill    75     11.50
Map Reader     Prentice-Hall 200     14.95

Title          Publisher     Qty     Price
```

To process **bookstock** and preserve tail data, the specification file **hi.fcl** contains:

```
/INFILE=bookstocktl
    /TAILREAD=88  # length of tail data
    /FIELD=(title, POSITION=1, SIZE=15)
    /FIELD=(publisher, POSITION=16, SIZE=13)
    /FIELD=(quantity, POSITION=30, SIZE=3, TYPE=NUMERIC)
    /FIELD=(price, POSITION=38, SIZE=5, TYPE=NUMERIC)
/REPORT
/OUTFILE=hiprice.out
    /INCLUDE WHERE price GT 10
    /TAILWRITE=bookstocktl # tail data
```

The resulting output, **hiprice.out**, is:

```
Reasoning For  Prentice-Hall 150     10.25
Still There    Dell           80     13.05
Sending Your   Valley Kill   130     15.75
People Please  Valley Kill    75     11.50
Map Reader     Prentice-Hall 200     14.95

Title          Publisher     Qty     Price
```

/RECSPERPAGE

This statement sets the number of records that are displayed before a footer and another header are written in **FieldShield** output. It is used in conjunction with `/FOOTREC` and/or `/HEADREC`. If `/RECSPERPAGE` is specified, the `/HEADREC` header and/or `/FOOTREC` footer will output on each page.

If the `/RECSPERPAGE` statement is not given, the header and/or footer will only be displayed once (at the start and end of the file, respectively).

Using an explicit value, for example:

```
/RECSPERPAGE=10
```

After every 10 records have been displayed or written, the header and/or footer will be repeated if `/HEADREC` and/or `/FOOTREC` are defined.

In order to force a page break, a "\f" should be included in the /FOOTREC statement.

U The statement used alone reads the user’s environment variable, `LINES`, and uses it for the value of `RECSPERPAGE`. Therefore, if `LINES` is set to 25, the header and/or footer will output every twenty five records. It would appear in the script as `/RECSPERPAGE`. ◆

3 ASCII/ALPHANUM Encryption and Decryption

This section describes the various encryption, decryption, masking, and hashing routines supplied with **FieldShield** to protect one or more ASCII or ALPHANUM fields / columns in your input source.

Advanced Encryption Standard (AES) ***on page 42***
Original field widths and field formatting are not preserved.

Format Preserving Encryption and Decryption (using AES 256) ***on page 45***
Original field widths are preserved, but not any original formatting.

Format and Type Preserving Encryption and Decryption for ALPHNUMERIC (using AES 256) ***on page 47***
Original field sizes and any original formatting are preserved.

GPG Encryption and Decryption ***on page 52***
Compatible with GPG key ring management. Original field widths and original field formatting are not preserved.

Triple Data Encryption Standard (3DES) ***on page 53***
All the original field widths and field formatting are not preserved.

All of the above encryption methods, except Format Preserving Encryption, produce printable ASCII characters (base64) as ciphertext.

The ssl and gpg cryptography transforms rely on third-party libraries to do the actual work. To use them, the appropriate third-party software must be installed on the same computer as the one running the IRI product. The library directory of the third-party software must be in the PATH, or the LD_LIBRARY_PATH for Unix type systems.

All other routines are provided in the library file **libcscrypt.so** or **libcscrypt.dll** (Windows). The GPG routines are provided in the file **libcsgpg.so** (UNIX/Linux) and **libcsgpg.dll** (Windows). The encode and decode for hex functions are provided in **libcsutil.so** (UNIX/Linux) and **libcsutil.dll** (Windows). The default location for all **FieldShield** library files is the **$FIELDSHIELD_HOME/lib** directory (UNIX/Linux) or ***install_dir*\lib** (Windows).

To use a library file, move or copy the file from **/lib** in the home directory to **/lib/modules**. If there are files in **/modules** that you are not using, it is recommended that you move the files back to **/lib**.

The **FieldShield** GPG library files have a dependency file, **cl32.so** (UNIX/Linux) or **cl32.dll** (Windows), which is provided with the product **cryptlib** (see Format Preserving Encryption and Decryption (using AES 256) *on page 45* for usage details).

NOTE

You can also use one or more external encryption routines instead of, or in addition to, the supplied routines. For complete details on writing and invoking your own custom field-level functions, see See Custom Masking Functions *on page 87*.

You can use multiple routines and multiple passphrases in the same **FieldShield** job script to vary protection among different input fields in order to improve security and target multiple recipients.

3.1 Usage: ASCII Encryption and Decryption

Encryption and decryption using the supplied **FieldShield** routines are supported only on fields declared as ASCII in the **FieldShield** job script.

NOTE

If you want to preserve a field's *original* data type for output display purposes (if other than ASCII), you can over-define the same field of data in the input section. This requires that you supply two distinct `/FIELD` statements for the same field of data in the `/INFILE` section of a job script -- for example, define one field named idnum_orig and declare its original data type, then define another field named idnum_to_encrypt (with the same SIZE and POSITION), where idnum_to_encrypt is declared as ASCII so it can be encrypted.

In the output section for display purposes, you can then specify the field idnum_orig with the desired output size and position, and declare its original data type (see Data Types (Single-Byte) *on page 159*).

Routines are invoked within a `/FIELD` statement in the `/OUTFILE` section of a job script.

3.2 Syntax: ASCII/ALPHANUM Encryption

The syntax for invoking an ASCII/ALPHANUM encryption or hashing routine is:

`routine_name(`*`field_name,passphrase_option`*`[,`*`public_kr_path] optional escape character`*`)`

where `routine_name` can be any of:

- enc_3des_ebc
- enc_3des_ssl
- enc_aes128
- enc_aes128_ssl
- enc_aes256
- enc_aes256_ssl
- enc_gpg
- enc_fp_aes256_alphanum
- enc_fp_aes256_alphanum_ssl
- enc_fp_aes256_ascii
- enc_fp_aes256_ascii_ssl
- hash_sha
- hash_sha2
- hash_sha2_ssl

Where *`field_name`* is the ASCII or ALPHANUM source field to be encrypted.

The *`passphrase_option`* can be any of the following:

"*literal_`string`*"	A user-supplied passphrase. Alternatively, you can specify this using `"pass:`*`literal_string`*`"`.
"file:[*path*/]*filename"*	For additional protection. A reference to a (restricted-access) file whose first line contains the passphrase. Any subsequent lines in that file are ignored. If the first line of that file is longer than 512 bytes, only the first 512 bytes are used. (Any trailing carriage returns or linefeeds are stripped from the passphrase line.)

NOTE

Environment variable substitution is supported for the literal string and `file:` options described above (see Environment Variables *on page 17*). However, you should use this option with caution because certain platforms allow you to view the environment of other processes (for example, when running certain UNIX operating systems).

No Argument	Applies only to `encryptAES256()`. If you do not supply a *`passphrase_option`*, a secret, internal passphrase is used.

	This passphrase is embedded inside the plug-in binary library, but is scrambled before use.
"***Key ID***"	Applies only to `enc_gpg()`. Supply the key ID that corresponds to the key inside the public key ring.

`public_kr_path` is a filename (with optional path) that contains the GPG public key ring. Used only with `enc_gpg()`.

NOTE

UNIX / Linux users can also apply the `setuid` file permission option to the **FieldShield** executable to help keep the passphrase secret, including from someone who has rights to execute the **FieldShield** script. This enables users to obtain access to files for which they do not have read permission, but certain fields will be encrypted.

The `file:` option described above is not required to accomplish this. It can be done using a literal string, but the `file:` methods provide more overall flexibility. Moreover, if the **FieldShield** job script itself is not protected, users can write their own scripts to decrypt the data.

3.3 Syntax: ASCII Decryption

After you have encrypted the values in one or more fields, the syntax for invoking the decryption routine in a subsequent job is:

```
routine_name(field_name,passphrase_option[,secret_kr_path])
```

where `routine_name` can be any of:

- dec_3des_ebc
- dec_3des_ssl
- dec_aes128
- dec_aes128_ssl
- dec_aes256
- dec_aes256_ssl
- dec_fp_aes256_alphanum
- dec_fp_aes256_alphanum_ssl
- dec_fp_aes256_ascii
- dec_fp_aes256_ascii_ssl
- dec_gpg

Where *`field_name`* is the source field to be decrypted, which must be the same *`field_name`* used in the previous encryption operation.

For decryption purposes, the `passphrase_option` is required as follows:

When using the `dec_gpg()` routine

Supply the password, in quotes, that corresponds to the key inside the secret key ring.

For all other decryption routines

The passphrase that you specified or referenced in the previous field-level encryption operation must be the same passphrase used for decryption. You can, however, use an alternative passphrase option. For example, you can use the literal string method for specifying the passphrase when encrypting, but use a `file:` option when decrypting, provided that the passphrase stored in the first line of the filename is identical to the literal passphrase given for the encryption operation. Similarly, when using `dec_aes256()`, if you used no argument for the *`passphrase_option`* for encryption, you must also use no argument for decryption.

`secret_kr_path` is a filename (with optional path) that contains the secret GPG key ring (see *Example 43* on page 120). Used only with `dec_gpg()`.

3.4 Advanced Encryption Standard (AES)

Standard encryption should be used when there is no requirement to preserve original field widths or any original field formatting.

The routines used for this method are `enc_aes256()` (for encryption) and `dec_aes256()` (for decryption).

AES encryption and decryption routines are also available using a 128-bit key algorithm. These routines are `enc_aes128` (for encryption) and `dec_aes128` (for decryption). All the rules applied to AES256 will also apply to AES128 routines.Standard Encryption

Example 12 Standard Encryption

Consider the following input file with three fields: date, credit card number, and amount:

```
10/12/2006 4111222233334444   21.30
10/12/2006 4555666677778888   19.99
10/13/2006 5999000011112222  256.00
10/15/2006 5222333344445555    8.77
10/15/2006 4111333355557777 3723.50
```

The following **FieldShield** script, **purchases_encrypt.fcl**, encrypts the credit card number field:

```
/INFILE=purchases.in
    /FIELD=(DATE, POSITION=1, SIZE=10, TYPE=AMERICAN_DATE)
    /FIELD=(CC, POSITION=12, SIZE=16)
    /FIELD=(TOTAL, POSITION=30, SIZE=7, TYPE=NUMERIC)
/REPORT
/OUTFILE=purchases.encrypted
    /FIELD=(DATE, POSITION=1, SIZE=10, TYPE=AMERICAN_DATE)
    /FIELD=(ENCC=enc_aes256(CC, "myPassword123"), POSITION=12, SIZE=24)
    /FIELD=(TOTAL, POSITION=37, SIZE=7, TYPE=NUMERIC)
```

The arguments passed to the `enc_aes256()` routine were `cc`, the field to be encrypted, and `myPassword123`, the encryption passphrase enclosed in quotes. This must be the same passphrase used for any subsequent decryption.

NOTE

As described in Syntax: ASCII/ALPHANUM Encryption *on page 39*, you can substitute the above literal passphrase `"myPassword123"` with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,
`"file:/home/usr/admin/Privatekey.txt"`
where the file **Privatekey.txt** contains `myPassword123` on the top line.

This produces **purchases.encrypted**:

```
10/12/2006 6jtwsg37heXMHQ1X7hbwiQ==   21.30
10/12/2006 2tpC7B4TSjoickODQ2MNPg==   19.99
10/13/2006 oftLzB4iNdmg7SDMZCsp5Q==  256.00
10/15/2006 XyCbgPOwSTNdOAl3N1qzMw==    8.77
10/15/2006 jLDqEg3s7Ct6R9kqlhaOAw== 3723.50
```

As shown above, the credit card number field has been obscured with printable ASCII characters and protected with 256-bit encryption.

Standard Encryption Field Size Considerations

In the previous example, note the size difference between the original credit card number field and the ciphertext output from encryption. Fields are encrypted in blocks of 16 bytes (characters). If a block is less than 16 bytes, it will be padded to the right with zero value bytes (0x00 or null).

Because the result of encryption is a binary value (may contain bytes in the range of 0 to 255), it is automatically encoded using the Base64 algorithm. Base64 replaces every 3 binary bytes with 4 printable characters. This results in a further

3:4 expansion of the field size. The steps below can be used to determine the size that **FieldShield** requires for the encrypted output field:

1) Round up the field size to the next even multiple of 16.
2) Divide by 3.
3) Round up to the next whole number.
4) Multiply by 4.

In the previous example, the original field size is 16. Because 16 is a multiple of 16, rounding up is not required. Therefore, 16 is divided by 3, and the result is rounded up to 6. 6 is then multiplied by 4 to arrive at the encrypted field size of 24.

NOTE

It is suggested that you specify the original field size when fields are to be decrypted (as shown in the output section of the script in the next example). An 8-character field will be padded with 8 null bytes to reach the 16-byte boundary. If the resultant 24-byte encrypted field is to be wholly decrypted, it will result in a 16-byte field that is padded with nulls.

Example 13 Standard Decryption

The next script, **purchases_decrypt.fcl**, decrypts the credit card numbers from the encrypted file **purchases.encrypted** that was produced in the previous example:

```
/INFILE=purchases.encrypted
    /FIELD=(DATE, POSITION=1, SIZE=10, TYPE=AMERICAN_DATE)
    /FIELD=(ENCC, POSITION=12, SIZE=24)
    /FIELD=(TOTAL, POSITION=37, SIZE=7, TYPE=NUMERIC)
/REPORT
/OUTFILE=purchases.restored
    /FIELD=(DATE, POSITION=1, SIZE=10, TYPE=AMERICAN_DATE)
    /FIELD=(CC=dec_aes256(ENCC, "myPassword123"), POSITION=12, SIZE=16)
    /FIELD=(TOTAL, POSITION=29, SIZE=7, TYPE=NUMERIC)
```

The arguments passed to the `dec_aes256()` routine were `encc`, the field to be decrypted, and `myPassword123`, which is the same passphrase used in the previous encryption operation, enclosed in quotes.

NOTE

As described in Syntax: ASCII Decryption *on page 41*, you can substitute the above literal passphrase `"myPassword123"` with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```
where the file **Privatekey.txt** contains `myPassword123` on the top line.

This produces **purchases.restored**, which restores the original credit card numbers:

```
10/12/2006 4111222233334444   21.30
10/12/2006 4555666677778888   19.99
10/13/2006 5999000011112222  256.00
10/15/2006 5222333344445555    8.77
10/15/2006 4111333355557777 3723.50
```

The `dec_aes256()` routine was applied to the credit card field, and the original numbers are restored.

3.5 Format Preserving Encryption and Decryption (using AES 256)

FieldShield supports the preservation of original field widths when encrypting ASCII fields. This is useful when you do not want to preserve custom format characteristics of a field, but need to retain original field widths.

The routines used for this method are `enc_fp_aes256_ascii()` (for encryption) and `dec_fp_aes256_ascii()` (for decryption).

Example 14 Width-Preserving Encryption

Consider the tab-delimited input file, **personal_info**, as shown in *Example 42* on page 119.

The following **FieldShield** script, **PII_ASC_enc.fcl**, encrypts the name field, while preserving its field widths:

```
/INFILE=personal_info
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE=personal_info_name_encrypted
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(ENC_NAME=enc_fp_aes256_ascii(NAME, "pass"), POSITION=3, \
            SEPARATOR="\t")
```

The arguments passed to the `enc_fp_aes256_ascii()` routine were the field name to be encrypted (`name`), followed by the encryption passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption.

The `enc_fp_aes256_ascii()` routine can be used with the ASCII. For the ASCII encryption you need to provide type=ASCII or, by default, ASCII.

`enc_fp_aes256_ascii()` routine is capable of taking three arguments where the first argument is a field name that is a mandatory argument for the function. This argument should not require quotes ("") around the field name. The second argument is a passphase enclosed in quotes ("pass"). This must be the same passphrase used for any subsequent decryption. This second argument is not mandatory. If you do not use a passphrase, then the function will use an internal salt as a passphrase argument. The third argument to be called is an escape character(s) that takes a single character or a string as an argument. These characters will be ignored in the encryption process. For example, if you have separators, such as a tab (\t) or comma (,), in your input data, then by mentioning ("\t,") this will ignore both the separator and the comma in the encryption process. This is how you preserve your separator characters from your input data.

NOTE As described in Syntax: ASCII/ALPHANUM Encryption *on page 39*, you can substitute the above literal passphrase `"pass"` with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,
`"file:/home/usr/admin/Privatekey.txt"`
where the file **Privatekey.txt** contains `pass` on the top line.

This produces **personal_info_name_encrypted**:

```
9654-4338-8732-8128    W389-324-33-473-Q    5 dGuN/a"Cp:"/I>
2312-7218-4829-0111    H583-832-87-178-P    j<K^m]~G.>
8940-8391-9147-8291    E372-273-92-893-G    HG.|HF0{*yq
6438-8932-2284-6262    L556-731-91-842-J     [xIcfp)#eg0
8291-7381-8291-7489    G803-389-53-934-J    D*9|I-G\o38w>
7828-8391-7737-0822    K991-892-02-578-O    SX0]ZAGfa7]
7834-5445-7823-7843    F894-895-10-215-N      PXdpCW0["a`,S8H
8383-9745-1230-4820    M352-811-49-765-N    n:gAG<fnN.1
3129-3648-3589-0848    S891-915-48-653-E    d=b\>" s\OB
0583-7290-7492-8375    Z538-482-61-543-M     :nMF^Q<)f
```

As demonstrated, the name field is encrypted, and the original field widths have been retained.

Example 15 Width-Preserving Decryption

The following script, **PII_ASC_dec.fcl**, decrypts the name field from the encrypted file **personal_info_name_encrypted** that was produced in the previous example:

```
/INFILE=personal_info_name_encrypted
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(ENC_NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE=personal_info_name_decrypted
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME=dec_fp_aes256_ascii(ENC_NAME, "pass"), POSITION=3, \
           SEPARATOR="\t")
```

The arguments passed to the `dec_fp_aes256_ascii()` routine were the field name to be decrypted (`name`), followed by the decryption passphrase enclosed in quotes (`"pass"`), which must be the same passphrase used in the previous encryption operation.

NOTE

As described in Syntax: ASCII Decryption *on page 41*, you can substitute the above literal passphrase `"pass"` with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,
`"file:/home/usr/admin/Privatekey.txt"`
where the file **Privatekey.txt** contains `pass` on the top line.

3.6 Format and Type Preserving Encryption and Decryption for ALPHNUMERIC (using AES 256)

FieldShield supports the preservation of both numeric and alphabetic characters (including upper- and lower-case letters) when encrypting fields, as well as their original field widths. Other characters, such as - or *, are left unaltered so that field formats such as those found in social security numbers and phone numbers are retained. This type of encryption is useful for fields with Personally Identifiable Information (PII), where the field formats must be preserved, but the values themselves must be encrypted.

The routines used for this method are `enc_fp_aes256_alphanum()` (for encryption) and `dec_fp_aes256_alphanum()` (for decryption).

Example 16 Alpha-Numeric Format-Preserving Encryption

Consider the tab-delimited input file, **personal_info**, as shown in *Example 42* on page 119.

The following **FieldShield** script, **PII_enc.fcl**, encrypts the credit card and driver's license number fields, while preserving the field formats:

```
/INFILE=personal_info
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE=personal_info_encrypted
    /FIELD=(ENC_CREDIT_CARD=enc_fp_aes256_alphanum(CREDIT_CARD, "pass"), \
           POSITION=1, SEPARATOR="\t")
    /FIELD=(ENC_DRIV_LIC=enc_fp_aes256_alphanum(DRIV_LIC, "pass"), \
           POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
```

The arguments passed to the `enc_fp_aes256_alphanum()` routines were the field names to be encrypted (`credit_card` or `driv_lic`), followed by the encryption passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption.

NOTE

As described in Syntax: ASCII/ALPHANUM Encryption *on page 39*, you can substitute the literal passphrase `"pass"` with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,
`"file:/home/usr/admin/Privatekey.txt"`
where the file **Privatekey.txt** contains `pass` on the top line.

This produces **personal_info_encrypted**:

```
0832-9678-1911-0645  R784-107-86-619-Q    Jessica Steffani
0835-7171-0577-5699  G156-454-45-303-O    Cody Blagg
0789-2128-0461-5374  Q305-118-71-384-Q    Jacob Blagg
1591-0561-0417-5772  D344-156-20-555-G    Just Rushlo
9296-9613-4710-5436  U751-860-67-075-Y    Maria Sheldon
9881-4436-0773-0973  X878-716-85-252-C    Keenan Ross
4594-9802-2566-4840  T273-579-67-063-M    Francesca Leonie
6514-3079-6147-6828  A617-849-83-864-X    Nadia Elyse
9221-6125-6496-9606  S039-406-12-369-U    Gordon Cade
1404-8512-8389-2619  K379-587-05-591-C    Hanna Fay
```

As shown above:

- The numeric values from the credit card number fields have been encrypted, but the field format remains intact. The encrypted values remain as digits, and the dashes have been retained.
- The alphabetic and numeric values from the driver's license fields have been encrypted, but the field format remains intact. The encrypted values remain as letters (upper-case preservation in this case) and digits where appropriate, and the dashes have been retained.

Example 17 Alpha-Numeric Format-Preserving Decryption

The next script, **PII_dec.fcl**, decrypts the credit card and driver's license number fields from the encrypted file **personal_info_encrypted** that was produced in the previous example:

```
/INFILE=personal_info_encrypted
    /FIELD=(ENC_CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(ENC_DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE=personal_info_decrypted
    /FIELD=(CREDIT_CARD=dec_fp_aes256_alphanum(ENC_CREDIT_CARD, "pass"), \
           POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC=dec_fp_aes256_alphanum(ENC_DRIV_LIC, "pass"), \
           POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
```

The arguments passed to the `dec_fp_aes256_alphanum()` routines were the field names to be decrypted (`credit_card` or `driv_lic`), followed by the decryption passphrase enclosed in quotes (`"pass"`), which must be the same passphrase used in the previous encryption operation.

NOTE

As described in Syntax: ASCII Decryption *on page 41*, you can substitute the above literal passphrase `"pass"` with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,
`"file:/home/usr/admin/Privatekey.txt"`
where the file **Privatekey.txt** contains `pass` on the top line.

Example 18 Using ASCII Encryption on Database Columns

As described in ODBC *on page 173*, you can use `/PROCESS=ODBC` in a **FieldShield** job script to process (on input) and populate (on output) table data in databases supported by Open Database Connectivity (ODBC).

This example demonstrates how the ASCII encryption and decryption routines can be applied to database data.Consider the following Oracle table, EMPLOYEE_INFO, the contents of which are displayed with a SQL SELECT query:

```
SELECT * FROM EMPLOYEE_INFO;
CREDIT_CARD         DRIV_LIC          NAME
------------------- ----------------- ------------------------------
9654-4338-8732-8128 W389-324-33-473-Q Jessica Steffani
2312-7218-4829-0111 H583-832-87-178-P Cody Blagg
8940-8391-9147-8291 E372-273-92-893-G Jacob Blagg
6438-8932-2284-6262 L556-731-91-842-J Just Rushlo
8291-7381-8291-7489 G803-389-53-934-J Maria Sheldon
7828-8391-7737-0822 K991-892-02-578-O Keenan Ross
7834-5445-7823-7843 F894-895-10-215-N Francesca Leonie
8383-9745-1230-4820 M352-811-49-765-N Nadia Elyse
3129-3648-3589-0848 S891-915-48-653-E Gordon Cade
0583-7290-7492-8375 Z538-482-61-543-M Hanna Fay

10 rows selected.
SQL>
```

The following script, **odbc_encrypt.fcl**, uses alpha-numeric format-preserving encryption for the credit card and driver's license number fields, and width-preserving encryption on the name field:

```
/INFILE="EMPLOYEE_INFO;DSN=oracle11g"  # identify source table and DSN
/PROCESS=ODBC                          # required for extracting RDBMS data
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE="EMPLOYEE_INFO;DSN=oracle11g" # identify target table and DSN
/PROCESS=ODBC                          # required for loading RDBMS data
/CREATE                                # clears existing rows first
    /FIELD=(CREDIT_CARD=enc_fp_aes256_alphanum(CREDIT_CARD,"pass"), \
            EXT_FIELD="CREDIT_CARD", POSITION=1,SEPARATOR="\t")
    /FIELD=(DRIV_LIC=enc_fp_aes256_alphanum(DRIV_LIC,"pass"), EXT_FIELD=DRIV_LIC, \
            POSITION=2,SEPARATOR="\t")
    /FIELD=(NAME=enc_fp_aes256_ascii()(NAME,"pass"), EXT_FIELD=NAME, \
            POSITION=3,SEPARATOR="\t"
```

The arguments passed to the `enc_fp_aes256_alphanum()` routines were the field names to be encrypted (`credit_card` or `driv_lic`), followed by the encryption passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption. The arguments passed to the `enc_fp_aes256_ascii()` routine were the field name to be encrypted (`name`), followed by the encryption passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption.

The use of `/PROCESS=ODBC` in the input section, along with the DSN and table name in the `/INFILE` statement ensure that data is accessed directly from the EMPLOYEE_INFO table in the oracle11g DSN. The use of `/PROCESS=ODBC` in the output section, along with the DSN and table name in the `/OUTFILE` statement ensure that the encrypted column data is loaded directly back into the EMPLOYEE_INFO table of the same DSN in this case. The `/CREATE` statement ensures that the original data rows are cleared before the new rows are inserted. For complete details on using `/PROCESS=ODBC`, see ODBC *on page 173*.

NOTE

When using `/PROCESS=ODBC` on output, you must use the `EXT_FIELD` option to name the original database source column name for each `/FIELD` that invokes a routine (see EXT_FIELD *on page 175*).

After execution, the database table, EMPLOYEE_INFO, contains the following encrypted data:

```
SELECT * FROM EMPLOYEE_INFO
CREDIT_CARD         DRIV_LIC          NAME
------------------- ----------------- ------------------------------
0832-9678-1911-0645 R784-107-86-619-Q 5 dGuN/a"Cp:"/I>
0835-7171-0577-5699 G156-454-45-303-O j<K^m]~G.>
0789-2128-0461-5374 Q305-118-71-384-Q HG.|HF0{*yq
1591-0561-0417-5772 D344-156-20-555-G [xIcfp)#eg0
9296-9613-4710-5436 U751-860-67-075-Y D*9|I-G\o38w>
9881-4436-0773-0973 X878-716-85-252-C SX0]ZAGfa7]
4594-9802-2566-4840 T273-579-67-063-M  PXdpCW0["a`,S8H
6514-3079-6147-6828 A617-849-83-864-X n:gAG<fnN.1
9221-6125-6496-9606 S039-406-12-369-U d=b\>" s\OB
1404-8512-8389-2619 K379-587-05-591-C :nMF^Q<)f

10 rows selected.
```

As shown above:

- The numeric values from the credit card number fields have been encrypted, but the field format remains intact. The encrypted values remain as digits, and the dashes were retained (see As described in Syntax: ASCII Decryption on page 41, you can substitute the above literal passphrase `"pass"` with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example, `"file:/home/usr/admin/Privatekey.txt"` where the file Privatekey.txt contains `pass` on the top line. *on page 47*).

- The alphabetic and numeric values from the driver's license number fields were encrypted, but the field format was preserved. The encrypted values remain as letters (upper-case preservation in this case) and digits where appropriate, and the dashes were retained.
- The name field is encrypted, but not with format preservation, and the original field widths were retained (see Format Preserving Encryption and Decryption (using AES 256) *on page 45*).

3.7 GPG Encryption and Decryption

GPG encryption and decryption routines can be used only when GPG is installed on your machine. You must use these routines if you want to apply GPG's key management facility to **FieldShield**'s field-level encryption and decryption operations.

NOTE

The GPG routines provided with **FieldShield** are also compatible with Pretty Good Privacy (PGP). The only difference between GPG and PGP is that GPG is a free product using a GNU General Public License (GPL), whereas PGP provides commercial support and it can be used for commercial purposes. Refer to your product license terms before generating keys with **FieldShield**'s GPG routines. Contact IRI if you need GPG/SSL (Windows only).

When using the GPG routines, you will be required to supply a public key ring for encryption, and a corresponding private / secret key ring for decryption (see Syntax: ASCII/ALPHANUM Encryption *on page 39* and Syntax: ASCII Decryption *on page 41*).

With GPG encryption, original field widths and any original field formatting are not preserved.

The routines used for this method are `enc_gpg()` and `dec_gpg()`. They are contained in the library file **libcsgpg.so** (UNIX/Linux) and **libcsgpg.dll** (Windows). By default, these are found in **$FIELDSHIELD_HOME/lib** (UNIX/ Linux) or ***install_dir*\lib** (Windows). The **FieldShield** GPG library files have a dependency file, **cl32.so** (UNIX/Linux) or **cl32.dll** (Windows), which is provided with the product **cryptlib**. You must move or copy the files from **/lib** in the home directory to **/lib/modules**. If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The files can also be referenced from the current working directory.

For examples of GPG encryption and decryption, see *Example 42* on page 119 and *Example 43* on page 120.

3.8 Triple Data Encryption Standard (3DES)

Use Triple DES encryption when there is no requirement to preserve original field widths or original field formatting.

The routines used for this method are `enc_3des_ebc ()` for encryption and `dec_3des_ebc ()` for decryption.

Example 19 Triple DES Encryption

Consider the following input file, **credit_card.in**, which contains credit card numbers.

```
8932-4338-8732-8128
2312-7218-4829-0111
8940-8391-9147-8128
6438-8932-2284-8291
8291-7381-8291-6262
9782-8391-7737-7489
7834-5445-7823-0822
8383-9745-1230-4820
3129-3648-3589-8128
0583-7290-7492-8375
```

The following FieldShield script, **credit_card_enc.fcl,** encrypts the numbers using 3DES encryption.

```
/INFILE=credit_card.in
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
/REPORT
/OUTFILE=credit_card_enc.out
    /FIELD=(ENC_CREDIT_CARD=enc_3des_ebc(CREDIT_CARD), POSITION=1, SEPARATOR="\t")
```

The first argument passed to `enc_3des_ebc()` is the field name, and the second argument is a passphrase.

This produces **credit_card_enc.out:**

```
mnO2HN0AERVajVP/Id6FJ0aJHmZ9Mrct
IsJc6CXXt6cdZrIpKcmTk9nndsAGC3h/
fMjp0ssGfgFzdz8U6in0aw3zya/1f9ZE
gqu1+XNfgjUs2ngER51M5u7lz8MOG22x
new0w0tFqBdh9PZuGqWHt4l9eeGlotj3
0s07u1SCMHm5tbbVQO543p5K1t4teA7W
1gOcOriaxkJr0V/GYzdXvHv9Ty96scGG
H6WfeXlKI8uRa4TSRd06ppT1IX1gY3/C
lnnT39HQt06nThp3RjuhlDKZJZkMZdoN
TI8nFF6fRMZRxkdEiXc03FtkkyKij41W
```

As shown above, the credit_card field has been obscured with printable ASCII characters and protected with 3DES encryption.

3DES Encryption Field Size Considerations

In the above example, note the size difference between the original input data and the encrypted output data. If the block size is less than 8 bytes, it will pad to the right with nulls. Because the result of encryption is a binary value, it is automatically enclosed using the base64 format. Every 3 bytes will be replaced with the 4 printable characters. This results in a further 3:4 expansion of the field size. The steps below can be used to determine the size that **FieldShield** requires for the 3DES output field:

1) Round up the field size to the next even multiple of 8.
2) Divide by 3.
3) Round up to the next whole number.
4) Multiply by 4.

Example 20 Triple DES Decryption

Consider the output file from the above example, **credit_card_enc.out,** as the input file.

The following **FieldShield** script, **credit_card_dec.fcl,** decrypts this input file. The results are the same as the original file, **credit_card.in**, that was encrypted in *Example 19*..

```
/INFILE=credit_card_enc.out
   /FIELD=(ENC_CREDIT_CARD, POSITION=1, SEPARATOR="\t")
/REPORT
/OUTFILE=credit_card_dec.out
    /FIELD=(CREDIT_CARD=dec_3des_ebc(ENC_CREDIT_CARD), POSITION=1, SEPARATOR="\t")
```

`dec_3des_ebc ()` is the field name and can pass two arguments. The first is the field name and the second is a passphrase. If there is no passphrase, **sortcl** uses an internal default passphrase.

This produces **credit_card_dec.out:**

```
8932-4338-8732-8128
2312-7218-4829-0111
8940-8391-9147-8128
6438-8932-2284-8291
8291-7381-8291-6262
9782-8391-7737-7489
7834-5445-7823-0822
8383-9745-1230-4820
3129-3648-3589-8128
0583-7290-7492-8375
```

4 ASCII De-identification and Re-identification

You can de-identify the contents of one or more output fields using an encryption-style *key* in cases where field data is of a sensitive nature and must be obscured. In a subsequent job, you can re-identify the field values to restore the original contents.

NOTE The algorithm for the FieldShield de-identification function is based on bit manipulation. It is not intended for encryption purposes. If you want to utilize Advanced Encryption Standard (AES) algorithms on your data, see ASCII/ ALPHANUM Encryption and Decryption *on page 38*.

The de-identification key used by **FieldShield** is a user-specified string, where the same string is used for subsequent re-identification, if required. The hexadecimal range of the data contents to be de-identified must fall within the ASCII printable character range.

The syntax for de-identification is:

```
/FIELD=(de_identify(field_name,"string"),other _attributes)
```

where `field_name` is the field to be de-identified, and `string` is any string that functions as the de-identification key.

The syntax for re-identification is:

```
/FIELD=(re_identify(field_name,"string"),other_attributes)
```

De-identifying Data *on page 105* and `string` is the same string that was used for de-identification.

See *Example 29* on page 105 and *Example 30* on page 106 for examples of de-identification and re-identification.

5 Encoding and Decoding

Use encoding routines when you need to encode byte data to be stored and transferred over media that are designed to deal with text data, such as email systems.

The routines used for this method are `encode_base64()`, `encode_base64_ssl()`, and `encode_hex()` for encryption.

`encode_base64()`

Converts the data in an ASCII string format into its base64 equivalent value. `encode_base64()` uses the encoding scheme as defined in RFC1113. Encoding causes an expansion of data. Every 3 bytes is replaced with 4 printable characters. This results in a further 3:4 expansion of the field size.

`encode_base64_ssl()`

Converts the data in an ASCII string format into its base64 equivalent value. `encode_base64_ssl()` uses the open ssl implementation of base 64 encoding. Encoding causes an expansion of data. Every 3 bytes is replaced with the 4 printable characters. This results in a further 3:4 expansion of the field size.

`encode_hex()`

Returns the hexadecimal representation of the byte input. Hex encoding converts 8 bit data to 2 hex characters. The hex characters are then stored as the two byte string representation of the characters. Hex encoding is probably a better choice than base64 encoding for human readability.

Use `decode_hex()`, `decode_base64()`, and `decode_base64_ssl()` to decode the above routines.

The encode and decode base64 routines are provided in the library file **libcscrypt.so** or **libcscrypt.dll** (Windows). The encode and decode for hex functions are provided in **libcsutil.so** (UNIX/Linux) and **libcsutil.dll** (Windows). The default location for all **FieldShield** library files is the **$FIELDSHIELD_HOME/lib/modules** directory (UNIX/Linux) or ***install_dir*\lib\modules** (Windows).

6 Hashing

The purpose of the (Secure Hash Algorithm) SHA-2 field-level routine is to return a hash of the data string in a given field or column.This is also known as a cryptographic hashing function. For any number of input bytes, the routine always returns a 32-byte hash code (in binary) that internally translates to a 44-byte, base64 value. Hashing is a useful technique for integrity checking.

SHA-1 is a 128-bit hash function that returns a 24 byte hash value. SHA-1 is less secure than SHA-2.

Example 21 Using SHA-2

Consider the following input file, **account_number.in**, which contains account numbers.

```
AWE-399732947327302
DKF-492376343732919
VMN-399732947327302
OIS-399732947327302
KAK-399732947327302
USK-399732947327302
KLS-399732947327302
DSJ-399732947327302
KWP-399732947327302
LSA-399732947327302
```

The following **FieldShield** script, **account_number.fcl**, takes the hash off the account numbers; the resulting hash will be in base64 format. The length of the base64 output will be 44 bytes of ASCII-readable characters.

```
/INFILE=account_number.in
    /FIELD=(ID_NUMBER, POSITION=1, SIZE=19, SEPARATOR="\t")
/REPORT
/OUTFILE=account_number.out
         # Specifies the target file (or DB target table)
    /FIELD=(H_ID_NUMBER=hash_sha2(ID_NUMBER), POSITION=1, SEPARATOR="\t")
```

The argument passed to `hash_sha2()` is the field name. The resulting data will be always a hash of the input field data.

This produces **account_number.out**:

```
MjIhEPbKMfh9/tv7KsqLAd/cXiIKac4KdBWVZBCNFEE=
FCKh0SUlq2PXvzdYy4ZKa9ZyRnfZ3obGx2kH+yybw/o=
g6vHVOO3LmJEUJ29qTD/mB9NMtW4Ax0q+UKQaC1WP2s=
uDx/P5A5DT1xElKWkFo1JQWPwFI7TICemIsIamz5Pdw=
tVXPjejWNJL4m5sRjjYNzhnqAV8MDBRf2SjXRd8QiR0=
Pa29aDcM+1ODXCzaiVeV/Y7IGJ9qcCaSIUo7ccLHsaE=
CiqaLjBz4Vd6+uZfwwToYKIzV6kexioy9uX8Zg9cKw0=
UkX5lwU+4DMajxr02qb92Wl6kqVWybFnky7KVDm3zL8=
BB3AmhSyn4Ya0jk3F7Gx3j3gK91cieZNVpyMK5405CY=
gjxv2Cuah6HSD4pEch8fUiv8sEBFRrDLy2atBxgYlW0=
```

7 String Masking (Redaction)

Data masking is used to preserve the original storage format and field appearance of data. In many cases, only specific parts of the data should remain exposed. For example, masking all but the last four digits of a credit card or social security number with asterisks is a common requirement.

7.1 Defining a Mask

7.1.1 Replace characters

To define a mask, use the `replace_chars` external field transform function with the following syntax:

```
/FIELD=(fieldname=replace_chars(field,"char",start,length),[other attributes])
```

where:

- *`fieldname`* is a name that is specified on input as the name of the new masked field.
- *`field`* is the name of the field to be masked.
- *`char`* is the character to be used for replacement.
- *`start`* is the numerical position at which the character replacement will start.
- *`length`* indicates the length of the piece to be replaced.

If the offset is a positive number, the offset is at the left of the string from which to start the mask. If the offset is a negative number, it signifes the offset is from the right, or end, of the string.

For example, to mask data values that are not consistent at the beginning, but are at the end, such as these European timestamps:

```
28/2/2014 13:12:22
15/4/2015 14:15:00
4/9/2010 02:36:01
7/11/2015 10:55:37
31/12/2014 23:59:59
```

You can use the following statement to mask the minutes and seconds:

```
/FIELD=(MASK_EURODATE=replace_chars(EURODATE, "0", -5, 2, "0", -2, 2)...
```

Resulting in:

28/2/2014 13:00:00

15/4/2015 14:00:00

4/9/2010 02:00:00

7/11/2015 10:00:00

31/12/2014 23:00:00

7.1.2 Format Strings

To provide formatted output, use the `format_strings` external field transform function with the following syntax:

```
/FIELD=( format_string [, field [, field [...]]] )
```

The first argument, `format_string`, is required. The format string is copied to the resultant field, but any string placeholders (%s) are replaced by the optional arguments, which can be ASCII fields or literal strings. However, any literal text can be given in the format string. Note that a beneficial use of this function is to create a field from a literal value, as shown in the example. The advantage of using the transform instead of a data statement is in the ability to name the entire field. This is important for loading database tables, and to be able to easily generate a DDF or DDL from an output section.

```
/INFILE=dummy
   /PROCESS=random
   /INCOLLECT=10
   /FIELD=(APPDATE, ISO_TIMESTAMP, SEPARATOR="\t", POSITION=2)
   /FIELD=(SSN3, DIGIT, SEPARATOR="\t", POSITION=3, SIZE=3)
   /FIELD=(SSN2, DIGIT, SEPARATOR="\t", POSITION=4, SIZE=2)
   /FIELD=(SSN4, DIGIT, SEPARATOR="\t", POSITION=5, SIZE=4)
/REPORT
/OUTFILE=format-strings-out.tab
   /FIELD=(APPOINTMENT_DATE=format_strings("%s.0", APPDATE), ASCII,\
      POSITION=1, SEPARATOR="\t")
   /FIELD=(SPONSOR_SSN=format_strings("%s-%s-%s", SSN3, SSN2,\
      SSN4), ASCII, POSITION=2, SEPARATOR="\t")
   /FIELD=(ACTIVE=format_strings("TRUE"), ASCII, POSITION=3, \
      SEPARATOR="\t")
```

The output is:

```
1914-05-12 02:01:46.0   120-85-4470 TRUE
1911-02-16 02:40:26.0   511-02-6901 TRUE
1958-08-19 09:26:35.0   299-89-4874 TRUE
1958-10-26 05:18:57.0   128-38-3620 TRUE
1929-01-15 03:32:00.0   595-73-5429 TRUE
1991-07-18 09:22:50.0   296-04-5023 TRUE
1952-10-14 00:22:33.0   417-72-4497 TRUE
1968-01-08 11:08:05.0   833-45-6650 TRUE
1935-07-14 09:53:08.0   865-38-1299 TRUE
1983-05-07 22:11:41.0   820-12-1609 TRUE
```

7.2 Using a Mask

You can replace a range of characters in a source field with a range of characters with a specified format of characters. Use a mask you define, or use one of the following formats:

Whole Field
```
/FIELD=(ID=replace_chars(id, "*"),POSITION=1,SEPARATOR=",")
```

Credit Card
```
/FIELD=(ID=replace_chars(id, "*", 1, 12),POSITION=28,SEPARATOR=",")
```

USA SSN (Social Security Number)
```
/FIELD=(ID=replace_chars(id, "*", 1, 3, "*", 5, 2),POSITION=40,SEPARATOR=",")
```

Canada SSN (Social Insurance Number)
```
/FIELD=(ID=replace_chars(id,"x",5,3,"x",9,3),POSITION=2,SEPARATOR=",")
```

Chile RUT (tax payer ID number)
```
/FIELD=(ID=replace_chars(id, "*", 2, 1, "*", 5, 2, "*", 9, 2, "*", 12, 1),\
POSITION=2, SEPARATOR=",")
```

Denmark CPR
```
/FIELD=(ID=replace_chars(id, "*", 3, 4, "*", 8, 3),POSITION=2,\
SEPARATOR=",")
```

Finland PIN
```
/FIELD=(ID=replace_chars(id, "*", 5, 2, "*", 8, 2, "*", 11, 1),\
 POSITION=2,SEPARATOR=",")
```

Hong Kong NID (National Identification)
```
/FIELD=(ID=replace_chars(id,"x"5,4),POSITION=2,SEPARATOR=",")
```

Iceland PIN
```
/FIELD=(ID=replace_chars(id, "*", 5, 2, "*", 8, 1, "*", 10, 2),\
POSITION=2,SEPARATOR=",")
```

Korea SSN (Social Security Number)
```
/FIELD=(ID=replace_chars(id,"x",1,6),POSITION=2,SEPARATOR=",")
```

Malaysia NRIC (National Registration Identity Card)
```
/FIELD=(ID=replace_chars(id, "*", 1, 2, "*", 5, 2, "*", 8, 2, "*", 14, 1),\
 POSITION=2,SEPARATOR=",")
```

Singapore NRIC
```
/FIELD=(ID=replace_chars(id, "*", 9, 1),POSITION=2,SEPARATOR=",")
```

Nordic Countries PIN/CPR (unique person identifiers)
```
/FIELD=(ID=replace_chars(id,"x",8,4),POSITION=2,SEPARATOR=",")
```

Spain DNI (National Identification)
```
/FIELD=(ID=replace_chars(id,"x",1,2,"x",4,3,"x",8,2),POSITION=2,\
 SEPARATOR=",")
```

Sweden PIN
`/FIELD=(ID=replace_chars(id, "*", 1, 2, "*", 5, 2, "*", 8, 2, "*", 11, 1),\`
`POSITION=2,SEPARATOR=",")`

Switzerland PIN
`/FIELD=(ID=replace_chars(id, "*", 7, 2, "*", 12, 2, "*", 15, 2),\`
`POSITION=2,SEPARATOR=",")`

Taiwan NIC (National Identification Card) `/FIELD=(ID=replace_chars(id, "*", 3, 4),POSITION=2,SEPARATOR=",")`
United Kingdom NINO (National Insurance Number) `/FIELD=(ID=replace_chars(id,"x",4,6),POSITION=5,SEPARATOR="|")`

7.2.1 Universally Unique Identifier

Use the Universally unique identifiers (UUIDs) transform function to assign unique identifiers to masked targets for obfuscating, de-identifying or auditing efforts. The syntax is:

```
/FIELD=(UUID=create_uuid()...
```

Which produces output similar to:

```
f4046e93-a393-4a2d-8d32-c389982c197b
d1856f14-62be-46f7-8e8b-4de2eb58c140
b13d53a8-5466-43ed-bc49-585a83e02848
478e07b7-f8d9-48b7-897b-be1c3752f91b
20975a92-5516-4893-8d7e-d82738718d96
```

You can create values that look like globally unique identifiers (GUIDs) by adding an optional parameter to create_uuid().

If a string literal is included, the function will use the first character as a frame, or quote, to wrap the UUID. If there are more than one characters in the string, the first and second characters will be used at the start and the end of the UUID respectively, as shown:

```
/FIELD=(UUID2=create_uuid("{}”))
```

This produces values that look like GUIDs:

```
{84323203-577a-439b-917c-ebbe239d2ed4}
{5648171d-2c66-4d78-b7de-dfa5f8a492c8}
{0e7b144f-4ddc-461b-bc9e-b71f93b4a0fd}
{01149109-391b-4aaa-8967-77728609570b}
{c0afb113-70fd-436a-a4f2-891e636c8b15}
```

8 Pseudonymization

When defining an output section of a script, **FieldShield** can substitute field values with other dependent or related field values. To display pseudonyms or aliases for specific input field values, a separate pre-sorted SET file (look-up file) is required for the substitutions, where the input data fields are paired with their pseudonyms. You can perform substitutions on every field for which a SET file is available.

The SET file must be arranged as rows of tab-delimited key-value pairs, where a unique value on the left will have an associated value on the right. Otherwise, different substitutions might be returned for the same look-up value.

9 SET Files

FieldShield can populate fields using set files in two different ways:

- where a value is searched for in the first column so that the value in a subsequent column is used for the field value (also known as a lookup). When a lookup is done, the search column must be ordered and the values in that column must be unique and in ascending order.
- where a value is randomly pulled from a column and then used as a field value. Single column set files always do a random pull.

NOTE

All SET files for use in **FieldShield** must be pre-sorted in ascending, alphabetical left-to-right in order and must have a line feed at the end of the last line.

You can generate a look-up table by extracting the relevant columns from a database, for example. The following is an example of SQL commands that will extract and sort two columns to a text-based, tab-delimited SET file:

```
SQL> spool name_code.set
SQL> select name||'  '||userid from personnel order by name;
SQL> spool off
```

where *name* is the first column containing the values to be looked-up and replaced by the values of their counterparts from the second column.

The space between the separators above (||''||) is a manually inserted tab.

9.1 Creating SET Files

Set files are composed of one or more columns. The columns must contain ASCII readable characters or EBCDIC characters and be separated by a tab. Each row or record ends with a linefeed. Commen ts can be placed at the top of a set file and are preceded by a # symbol. The set records begin with the first line that is not preceded by a # symbol. Thereafter, if a # appears, it is assumed to be part of the data.

Set files can be created from a number of different sources and use any one of a number of techniques. They can be created with a text editor; they can be extracted from a database using a database procedure or by using a script. A set file can also be created from other data sources. By convention, set files are given a file name with the .set extension.

You can create any of the following types of set files:

9.1.1 Character

A character set file must have values that are ASCII readable.

9.1.2 Numeric

A numeric set file can have any combination of numbers and numeric ranges for the entries. A range has a lower limit and an upper limit separated by a comma and enclosed with brackets. A square bracket indicates that the limit next to the bracket is to be included in the range and a rounded bracket indicates that the limit is not to be included. Below is an example of a numeric set file.

```
(-100,-10]
-2
0
2
[10,100)
[10,100)
```

First is a random pull from the rows, then from within the range in a row. Notice that the range [10,100) is in the set twice. Since there are 6 rows in the set file, approximately 1/3 of the values will be from that range, while there will be values from each of the other rows approximately 1/6 of the time. Values in the [10,100) range can include the number 10 but not the number 100.

9.1.3 Date

A date set file can have any combination of dates and date ranges for the entries. The ranges are done in the same form as numeric ranges using iso_date, iso_time, or iso_timestamp formats. These can then be converted to other TYPES.

9.1.4 Multi-column

Multi-column set files can be used for random pulls or for lookups.

- When doing an ordinary lookup, the first column must be sorted and unique for a 2-column set file. For a 3-column set file, the first column is ordered and the second column is ordered and unique with reference to the first column value, and so on for more columns.
  A 2-column set file with states and capitals could look like this:

GA	Atlanta
FL	Tallahassee
MS	Jackson

  A 3-column set file with product, price_type, and amount could look like this:

hammer	cost	5.95
hammer	retail	10.95
hammer	sale	8.95
tablesaw	cost	185.00
tablesaw	retail	249.00
tablesaw	sale	219.25

- When doing a random pull, the items in a row are usually related so that some or all of the related items can go into the same data record. You could have a 2-column set file that has states and cities that are in the state of the first column. It is not necessary for the columns to be ordered because you are randomly picking the row and then using the values from the columns in that row.
- When doing a random lookup, the first column is ordered, but not unique. The second column will be related to the first column. Therefore, the value in the first column is used to do a random pull from the related items in the second column. One example of this is to have states ordered with related cities in the second column. Then the state value in the data record is used to pull a random city for that state.

9.2 Using SET Files in a Job Script

Sets are invoked in the `/FIELD` statement with the following syntax.

```
/FIELD=(FieldName, FieldAttributes, SET="<Set_Source>" [<[SearchList]>]
DEFAULT="string" [ORDER=<Order Option>] [SELECT=<Select Option>]
[SEARCH=<Search Option>] )
```

where:

- *`FieldName`* is the name of the field for the record that is being processed.
- *`FieldAttributes`* are the attributes that are independent of what is done to describe the set processing, such as POSITION, SIZE, and data type.
- *`SET="Set_Source"`* is one of the following:
  - the path and file name for the set file. This must have quotes around it.
  - a list of values separated by commas that are used in place of a set file. These values are enclosed in curly brackets { value1, value2, . . }. This is normally used when there are only a few values, as in:

```
/FIELD=(response, POSITION=2, SEPARATOR="\t", SET={ Y, N })
```

- *`SearchList`* is a list of comma separated values that is used for a lookup. There can be up to one less than the number of columns in the set file. The items in this list can be a field name or a literal and are separated by commas and enclosed in square brackets [item1, item2, . . .]. This is also used for a random lookup where each column in the set file is ordered and the next to last column has the value that determines the random pull for the last column. You can use the select options ANY, ALL, ONCE, and SUFFIX with a random lookup. In the following example set file, the first column is state, the second is county, and the third is city.

CA	San Bernardino	Highland
CA	San Bernardino	San Bernardino
FL	Brevard	Melbourne
FL	Brevard	Titusville
FL	Hillsborough	Tampa

To get random cities within each state and county combination, you could use the following job script.

```
/INFILE=rlook.dat
    /FIELD=(id,POSITION=1,SEPARATOR="|")
    /FIELD=(state,POSITION=2,SEPARATOR="|")
    /FIELD=(county,POSITION=3,SEPARATOR="|")
/REPORT
/OUTFILE=rlook.out
    /FIELD=(id,POSITION=1,SEPARATOR="|")
    /FIELD=(state,POSITION=2,SEPARATOR="|")
    /FIELD=(county,POSITION=3,SEPARATOR="|")
    /FIELD=(city,POSITION=4,SEPARATOR="|",\
         SET=”state_county_city.set” [ state,county ])
```

When executed, this job script results in similary output to:

```
001|CA|San Bernardino|Highland
002|FL|Hillsborough|Tampa
003|CA|San Bernardino|San Bernardino
004|FL|Brevard|Melbourne
005|FL|Hillsborough|Tampa
006|FL|Brevard|Melbourne
007|FL|Brevard|Melbourne
008|FL|Brevard|Titusville
```

The input contains three fields that are mapped to output. The output also includes the lookup for city, based on state and county.

- *DEFAULT="string"* is the value placed in the data field when a lookup search fails to find a match in the set file. If DEFAULT is not set, an error message results when there is no match for the lookup, and execution of the script stops.
- *ORDER=Order Options* is used when the set file is being used as a lookup. A set must be ordered for a lookup to work correctly. When you do an ordinary lookup, the first column must be sorted and unique for a 2-column set file. For a 3-column set file, the first column is ordered and the second column is ordered and unique, with reference to the first column value, and so on for more columns.
  - *PRE_SORTED* assumes that the set file is already ordered correctly to do a lookup with no duplicates for the appropriate column. This is the default setting.
  - *NOT_SORTED* means the set file will be sorted internally prior to processing the job script. The sorted set file will not be saved.

- `SELECT=Select Options` is used for random pulls.

`ANY`	Values from the set file are randomly pulled from any row of the set file. Values that are pulled can be pulled any number of times. This is the default.
`ALL`	Values are pulled from the rows of the set file sequentially. All rows will be used for values if the data file has a sufficient number of records to do this. If there are more data records than rows in the set file, the values being pulled from the set file will be repeated.
`ONCE`	Values are pulled from the rows of the set file sequentially starting at the top row of the set file. When all the rows have been used to pull values, then no more values from the set file will be inserted into the field in the data file and that field will be empty for the remaining records.
`SUFFIX`	This behaves similar to ALL. Values are pulled from the rows of the set file sequentially. If there are more data records than rows in the set file, when the values in the set file repeat, an underscore and numeric values in sequence will be added to the value in the data file, such as John_2, John_3, John_4, and so on.
`ROW`	Only applies to set files with two or more tab delimited columns. Works like ANY if the referenced set file has not been used in a previous field. If the referenced set file has already been used in a previous field, then the same row from the set file is used to supply the value for the field with the ROW selection type. The required index argument to the ROW selection type specifies which column of the set file to use to supply the field value. You can also use ROW to do a random pull from any one of the set columns. Here is an example of three fields that use a set file that has state, county, and city columns respectively.

```
/FIELD=(city, POSITION=2, SEPARATOR=',', SET= 3column.set SELECT= ROW[3])
/FIELD=(county, POSITION=3, SEPARATOR=',', SET= 3column.set SELECT=ROW[2])
/FIELD=(state, POSITION=4, SEPARATOR=',', SET= 3column.set SELECT= ROW[1])
```

Those field definitions could yield records with values such as this:

```
COLUMBUS,LOWNDES,MS
TAMPA,HILLSBOROUGH,FL
```

You can also use ROW to do a random pull from any one of the set columns. If you only want to have random counties in the record, use a field statement similar to the following: `/FIELD=(county, POSITION=3, SEPARATOR=',', SET= 3column.set SELECT= ROW[2])`

```
/FIELD=(county, POSITION=3, SEPARATOR=',', SET= 3column.set \
SELECT= ROW[2])
```

PERMUTE This is used in the input section of a script that has `/PROCESS=RANDOM`. For each field that has PERMUTE as the selection parameter for a set file, there will be an attempt to select all possible combinations for those set files. This is limited by the number put in the `/INCOLLECT` statement. If you want only all combinations, `/INCOLLECT` can be set to PERMUTE. If `/INCOLLECT` is set to a number greater than all the combinations, the combination pattern will repeat.

WEIGHT Applies only to weighted distributions. The Weighted Distribution requires the existence of a set file with two or more entries with five parameters that must be tab separated. The parameters are percentage, beginning minimum value, ending minimum value, beginning maximum value, and ending maximum value. The percentages in the first column must add up to 100.

- *`SEARCH=Search Options`* The set searches that have been previously covered have been based on finding exact matches. The options here will also describe how to use set files that use operators to obtain unequal values. One application for using the unequal values is with Slowly Changing Dimensions..

  EQ The search parameter is looking for an exact match in the set file. This is the default search option.

  GT The search parameter is looking for the first value in the set file that is greater than the search parameter.

  GE The search parameter is looking for an exact match. If an exact match is not found, then the first value encountered that is greater than the search parameter is selected.

`LT`	The search parameter is looking for the first value in the set file that is less than the search parameter.
`LE`	The search parameter is looking for an exact match. If an exact match is not found, then the first value encountered that is less than the search parameter is selected.

Example 22   Substituting Sensitive Data Values (Pseudonymization)

This example uses table look-ups to substitute the values of one or more fields in order to prevent the display of sensitive data. You can subsequently restore the substituted values with their original contents.

NOTE

See Randomization *on page 72* for details on the field functions `de_identification` and `re-identification` that are used for the random jumbling and subsequent restoring of field values based on an encryption-style key. The SET file method of substituting values described in this example uses a look-up table to restore the original field contents.

This example shows how you can substitute sensitive field values and create a look-up table in one job script, and then restore the original values by referring to this look-up table in a subsequent job script.

Consider the following input data, **Sensitive**, which contains names and salary figures:

```
Jones|Alan|125000
Smith|Abraham|67000
Guiness|Michael|122650
Schwartz|Francis|78234
Jones|Rick|67234
Williams|Billie|87342
```

The following script, **substitute.fcl**, produces three output files -- one that substitutes values from the lname and fname fields, and two that create look-up tables to be used to restore the lname and fname fields in a subsequent job:

```
/INFILE=Sensitive # input file with sensitive data
  /FIELD=(LNAME, POSITION=1, SEPARATOR="|")
  /FIELD=(FNAME, POSITION=2, SEPARATOR="|")
  /FIELD=(SALARY, POSITION=3, SEPARATOR="|")
/REPORT
/OUTFILE=Operational # substituted data file
  /FIELD=(SEQUENCER, POSITION=1,SEPARATOR="|")
  /FIELD=(NewLNAME, POSITION=2, SEPARATOR="|", SET="names_last.set")
  /FIELD=(NewFNAME, POSITION=3, SEPARATOR="|", SET="names_first.set")
  /FIELD=(SALARY, POSITION=4, SEPARATOR="|")
/OUTFILE=fname.set # used to restore later
  /FIELD=(SEQUENCER, POSITION=1, SEPARATOR="\t")
  /FIELD=(FNAME, POSITION=2, SEPARATOR="\t") # tab-delimited
/OUTFILE=lname.set # used to restore later
  /FIELD=(SEQUENCER, POSITION=1, SEPARATOR="\t")
  /FIELD=(LNAME, POSITION=2, SEPARATOR="\t") # tab-delimited
```

This produces **Operational**:

```
1|McGurk|Ira|125000
2|Stirling|Justina|67000
3|Sproles|Alexandria|122650
4|Fackler|Arden|78234
5|Vila|Bertram|67234
6|Isakson|Georgianna|87342
```

It also produces the tab-delimited SET files **lname.set:**

```
1     Jones
2     Smith
3     Guiness
4     Schwartz
5     Jones
6     Williams
```

and **fname.set**:

```
1     Alan
2     Abraham
3     Michael
4     Francis
5     Rick
6     Billie
```

The output file **Operational** can be viewed and analyzed by those without the authority to view real names. The sequenced values on the left represent records with actual names, and look-up values are kept in **lname.set** and **fname.set**.

SEQUENCER was used as a field name to produce unique, incrementing values (see SEQUENCER *on page 147*) in both output files. These values can then be matched across the files **Operational** and the created set files, as illustrated in the following script, **restore.fcl**.

NOTE

This example shows one method for generating a look-up table. DBMS users may wish to export tab-delimited tables to be used as SET files, for example.

The next script, **restore.fcl**, restores the salary figures to their original names using table look-up functionality:

```
/INFILE=Operational
   /PROCESS=RECORD
   /ALIAS=Operational
   /FIELD=(CODE, TYPE=ASCII, POSITION=1, SEPARATOR="|")
   /FIELD=(FAKE_LNAME, TYPE=ASCII, POSITION=2, SEPARATOR="|")
   /FIELD=(FAKE_FNAME, TYPE=ASCII, POSITION=3, SEPARATOR="|")
   /FIELD=(SALARY, TYPE=NUMERIC, POSITION=4, SEPARATOR="|", PRECISION=2)
/REPORT
/OUTFILE=restored.out
   /FIELD=(CODE, TYPE=ASCII, POSITION=1, SEPARATOR="|")
   /FIELD=(LNAME, TYPE=ASCII, POSITION=2, SEPARATOR="|", SET="lname.set" [code])
   /FIELD=(FNAME, TYPE=ASCII, POSITION=3, SEPARATOR="|", SET="fname.set" [code])
   /FIELD=(SALARY, TYPE=NUMERIC, POSITION=4, SEPARATOR="|", PRECISION=2)
```

The name field on output returns the look-up equivalent to the code value found in the SET file.

This produces **Restored**:

```
1|Jones|Alan|125000
2|Smith|Abraham|67000
3|Guiness|Michael|122650
4|Schwartz|Francis|78234
5|Jones|Rick|67234
6|Williams|Billie|87342
```

The original names have been restored based on the codes found in the set files.

For an example of using look-up tables for pseudonymization purposes, see *Example 33* on page 109.

10 Tokenization

Tokenization is a method of protecting valuable information with a token that masks the original value. The advantage to protecting information this way is that it cannot be reverse engineered. Encryption uses a set of rules to create a value that will represent the input value. These rules can be traced and reversed to get the original number. With tokenization, the original value is changed base on an index, a random static number, or through a combination of both. Doing this results in the information being stored in a database or file that only specific users can access.

Tokenization is widely used to mask credit card numbers. The Payment Card Industry Data Security Standard, or PCI DSS, has a set of rules that a company must comply to in order to be compliant and tell its customers that it does safely store their payment card information.

In FieldShield, tokenization works through the use of an external facility. Contact IRI or your IRI agent for more information.

11 Randomization

The three types of random SET files used in **FieldShield** are:

- Alpha-numeric list of values
- Numeric values and ranges
- Date values and ranges

11.1 Alpha-numeric list of values

Field values can be drawn at random from a pre-existing SET file that contains any number of alpha-numeric values. When you are selecting values from a SET file, the values appear in your output exactly as they appear in the SET file. This provides realistic-looking data.

11.2 Numeric values and ranges

FieldShield can also draw from numeric values at random from pre-existing SET files that contain any number of numeric values or numeric ranges. The SIZE attribute you provide determines the precision of how the value will be represented (unless a literal value is selected). When you are selecting values from a numeric SET file, the NUMERIC data type should be assigned to that field.

The supported entries in a numeric SET file can be any combination of:

- Literal values with or without precision, such as 14 or 12.25. These values are produced as they appear.
- [x,y]. Inclusive low to high range values, where x and y are considered for random selection. For example, the entry [-2,2] can yield any of the following (if decimal precision is set to 0):
    - -2
    - -1
    - 0
    - 1
    - 2
- (x,y). Exclusive low to high range values, where x and y are not considered for random selection, but all values in between are considered. For example, the entry [-2,2] can yield any of the following (if decimal precision is set to 0):
    - -1
    - 0
    - 1
- [x,y] or [x,y]. A combination pair of square and round brackets where the above rules apply to each side of the expression. For example, the entry [-2,1] can yield any of the following (if decimal precision is set to 0):
    - -2
    - -1
    - 0
- Numeric ranges using decimal precision. For example, the entry [1.52,1.55] can yield any of the following (if decimal precision is set to 2):
    - 1.52
    - 1.53
    - 1.54
    - 1.55

11.3 Date values and ranges

FieldShield can also draw from dates at random from pre-existing SET files that contain any number of date values or ranges. Only valid dates are produced when date ranges are selected from a SET file.

Literal date values in a SET file are also ignored by FieldShield when they are invalid. For example, an entry of Feb/31/2005 is ignored.

When you are selecting values from a date SET file, a FieldShield-supported DATE data type must be assigned to that field (see *Table 2*) and the dates within the SET file must be of that specified data type. Note also that the SIZE attribute you provide must correlate to the width of date value.

Table 2: Date Formats

Variable	Format	Example
AMERICAN_DATE	*month/day/year*	Dec/31/2010 or 12/31/2010
EUROPEAN_DATE	*day.month.year*	31.12.2010 or 31.Dec.2010
JAPANESE_DATE	*year-month-day*	2010-12-31 or 2010-Dec-31
ISO_DATE	*year-month-day*	2010-12-31 or 2010-Dec-31

The supported entries in a date SET file can be any combination of:

- Literal date values, of a FieldShield-supported DATE data type, such as 07/31/2004 or 2004-12-31.
- [x,y]. Inclusive low to high range values, where x and y are considered for random selection. For example, the entry [2002-02-27,2002-03-02] can yield any of the following:
  - 2002-02-27
  - 2002-02-28
  - 2002-03-01
  - 2002-03-02
- [x,y]. Exclusive low to high range values, where x and y are not considered for random selection, but all values in between may be generated. For example, the entry (2002-02-27,2002-03-02) can yield either of the following:
  - 2002-02-28
  - 2002-03-01
- [x,y] or [x,y]. A combination pair of square and round brackets where the above rules apply to each side of the expression. For example, the entry [2002-02-27,2002-03-02) can yield any of the following:
  - 2002-02-27
  - 2002-02-28
  - 2002-03-01

12 /Re-formatting and Masking Options

`/DATA` statements can be used to pad, mask, and format output records. `/DATA` statements are not named fields, so they cannot be directly mapped. They are positioned just after the previous`/DATA` or `/FIELD` statement.

There are several forms of `/DATA` statements:

`/DATA=[{n}]"literal_string"`

Where *n* is the number of times to repeat the `literal_string` that you specify, and the constant string can contain any combination of constants and control characters (see Control (Escape) Characters *on page 75*).

NOTE The repetitive string syntax using `{n}"literal_string"` is also supported in a `/CONDITION` statement, and can be used for both record-level and field-level evaluation purposes (see Binary Logical Expressions *on page 163*).

`/DATA=field_name`

Displays the value of an input field without formatting.

`/DATA=internal_variable`

Where the internal variable can be any **FieldShield** internal variable (see Internal Variables *on page 76*)

You can also use a conditional `/DATA` statement (see Conditional Field Security and Masking *on page 19*).

Control (Escape) Characters

Statements using `/DATA`, `/HEADREC`, and `/FOOTREC` can also contain control characters. These characters, shown in the following table below, can be used to produce special effects in your output records (see *Example 23* on page 77).

Table 3: Control (Escape) Characters

Charac-ter	Format within a String
`\a`	alert
`\\`	backslash
`\b`	backspace
`\r`	carriage return
`\"`	double quote
`\f`	form-feed
`\t`	horizontal tab

Table 3: Control (Escape) Characters

Charac-ter	Format within a String
`\n`	newline
`\0`	null character
`\'`	single quote
`\v`	vertical tab
`\$`	dollar sign

Internal Variables

FieldShield maintains internal values you can use in `/DATA` statements for additional record formatting in your target file(s). The internal values are shown in the following table.

Table 4: Internal Variables

Variable	Output/Example

`AMERICAN_DATE`
`EUROPEAN_DATE`
`JAPANESE_DATE`
`ISO_DATE`
`CURRENT_DATE`
	`Sep/19/2006`
	`19.09.2006`
	`2006-09-19`
	`2006-09-19`
	`2006-09-19`

`AMERICAN_TIME`
`EUROPEAN_TIME`
`JAPANESE_TIME`
`ISO_TIME`
`CURRENT_TIME`
`VALUE_TIME`
	`09:47:15 AM`
	`21.47.15`
	`9:47:15 PM`
	`21:47:15`
	`21.47.15`
	`350:47:15`

`AMERICAN_TIMESTAMP`
`EUROPEAN_TIMESTAMP`
`JAPANESE_TIMESTAMP`
`ISO_TIMESTAMP`
`CURRENT_TIMESTAMP`
`VALUE_TIMESTAMP`
	`Sep/19/2009 09:47:15 AM`
	`19.09.2009 21.47.15`
	`2009-09-19 9:47:15 PM`
	`2009-09-19 21:47:15`
	`2009-09-19 21.47.15`
	`247 350:47:15`

`SYSDATE`	current date and time in the format used by Oracle and SQLbase
`CURRENT_TIMEZONE`	`Eastern Daylight Time` (Windows) `EDT` (UNIX)

Table 4: Internal Variables

Variable	Output/Example
`PAGE_NUMBER` (`/HEADREC` and `/FOOTREC` statements only)	page number of the report
`USER`	the name of the user currently executing the program

For example, if you include the following statement in your script:

```
/HEADREC="Produced on %s",CURRENT_DATE
```

then the top of the output might contain:

```
Produced on 2006-09-19
```

Or, if you include the following statement in your script:

```
/DATA=CURRENT_DATE
```

then each record will contain, for example:

```
2009-01-27
```

Example 23 Using /DATA Statements

Consider the input file **miami**:

```
Reasoning For  Prentice-Hall 150     10.25
Murder Plots   Harper-Row    160      5.90
Still There    Dell           80     13.05
Pressure Cook  Harper-Row    228      9.95
Sending Your   Valley Kill   130     15.75
People Please  Valley Kill    75     11.50
Map Reader     Prentice-Hall 200     14.95
```

The following script, **miami.fcl**, includes several instances of `/DATA` statements:

```
/INFILE=miami
    /FIELD=(Title, POSITION=1, SIZE=15)
    /FIELD=(Publisher, POSITION=16, SIZE=13)
/REPORT
/OUTFILE=miami_report.out
    /FIELD=(Publisher, POSITION=1, SIZE=15)
    /DATA="Book: "  #constant string
    /DATA=Title     #field-name
    /DATA=" "       #constant string
    /DATA=CURRENT_TIMESTAMP  #internal variable
    /DATA={3}"*-*"  #repeated constant string
    /DATA="\n"      #linefeed control character
```

This produces:

```
Prentice-Hall  Book: Reasoning For   2012-09-19 12:51:34*-**-**-*

Harper-Row     Book: Murder Plots    2012-09-19 12:51:34*-**-**-*

Dell           Book: Still There     2012-09-19 12:51:34*-**-**-*

Harper-Row     Book: Pressure Cook   2012-09-19 12:51:34*-**-**-*

Valley Kill    Book: Sending Your    2012-09-19 12:51:34*-**-**-*

Valley Kill    Book: People Please   2012-09-19 12:51:34*-**-**-*

Prentice-Hall  Book: Map Reader      2012-09-19 12:51:34*-**-**-*
```

The extra space between records was produced by the statement `/DATA="\n"`.

See *Example 34* on page 111 for an example of using a repeated constant string for the purposes of masking with `/DATA` statements.

12.1 Find and Replace

The find and replace feature provides another way to mask or obfuscate sensitive data. It produces a derived field value in the output record by examining the source field and replacing every occurrence of a specified search string with the specified replace string. Both the search and replace string can be a field value or a literal string, and you can use find and replace statements for one or more output fields.

The syntax is:

/FIELD=(find_and_replace(*source_field*,*search_string*,*replace_string* or *field_name*),*other_attributes*)

For example, consider an input field that is defined as follows:

```
/FIELD=(Pres,POSITION=1,SIZE=25)
```

If the value of Pres is:

```
Kennedy
```

then the value produced to `/OUTPUT` is as follows, depending on what you specify:

`FIND_AND_REPLACE(Pres,"en","EN")` will produce `KENnedy`

`FIND_AND_REPLACE(Pres,"en","1234")` will produce `K1234nedy`

`FIND_AND_REPLACE(Pres,"e","")` will produce `Knndy`

If the value of the field Pres2 is Johnson, while the value of Pres is Kennedy, for example, you could specify:

`FIND_AND_REPLACE(Pres,"K",Pres2)` which will produce `Johnsonennedy`.

NOTE

If the intended find or replace value is a double quote ("), then you must escape the string itself with a backslash, for example:

```
/field=(find_and_replace(f2,"\"","*"),position=1, \
separator=',',frame='"')
```

Find and Replace: 2string Functions

With 2string functions, identify one or more characters within a field that does not conform to a specified criterion, and replace each of the non-conforming characters with a user-defined string, or with the value of another field from the record.

The 2string function is applied in the `/INREC` or output record. The find string is one of several iscompare-style conditions, and the replace string can be a literal string or a field value. The available 2string functions are:

`non_print2s(`*`field1`*`,"`*`string`*`" or `*`field2`*`)`
: Replaces all non-printable characters in *`field1`* with the specified string or value from *`field2`*.

`non_alpha2s(`*`field1`*`,"`*`string`*`" or `*`field2`*`)`
: Replaces all non-alphabetic characters (in the current locale) in *`field1`* with the specified string or value from *`field2`*.

`non_alnum2s(`*`field1`*`,"`*`string`*`" or `*`field2`*`)`
: Replaces all non-alphabetic characters (in the current locale) and non-digits in *`field1`* with the specified string or value from *`field2`*.

`non_ascii2s(`*`field1`*`,"`*`string`*`" or `*`field2`*`)`
: Replaces all non-7-bit unsigned char values (from the ASCII character set) in *`field1`* with the specified string or value from *`field2`*.

`non_digit2s(`*`field1`*`,"`*`string`*`" or `*`field2`*`)`
: Replaces all non-digits in *`field1`* with the specified string or value from *`field2`*.

`non_asc_let2s(`*`field1`*`,"`*`string`*`" or `*`field2`*`)`
: Replaces all non-ASCII characters in *`field1`* with the specified string or value from *`field2`*.

`non_ebc_let2s(`*`field1`*`,"`*`string`*`" or `*`field2`*`)`
: Replaces all non-EBCDIC characters in *`field1`* with the specified string or value from *`field2`*.

The syntax is, for example:

```
/FIELD=(non_print2s(source_field,replace_string or field_name),POSITION=...)
```

NOTE

The list of printable characters on Windows operating systems differs from printable characters on Linux and Unix. Refer to your operating system documentation for details.
Be sure to size the 2string field appropriately to accommodate a larger field if one or more characters may be replaced by a string of more than one byte.

Find and Replace: Case Transfer Functions

The following case transfer functions operate on the alphabetic characters of an ASCII string. The string can be a either a field value or a quoted literal. The incoming case is not considered. The case transfer functions are as follows:

`toupper(`*`S`*`)` The resultant string will contain all upper-case letters.

`tolower(`*`S`*`)` The resultant string will contain all lower-case letters.

`toproper(`*`S`*`)` The resultant string will contain both upper- and lower-case letters.

The following are converted to upper-case:

- the first character of *`S`*

- any character that follows an apostrophe (')
- any character that follows Mc
- any character that follows a tab or other white space character.

All other characters will be converted to lower case.

The syntax is, for example:

```
/FIELD=(toupper(field or "string"),POSITION=...)
```

where *field* is the field name whose contents will be converted, and "*string*" is a literal string value to be converted.

12.2 Named Fields With a Literal (fixed) Value

You can assign a literal string or functional value to a named field in input, inrec, or output.

A literal value is a numeric value or a quoted string; a numeric value can also be an expression enclosed in parenthesis. The field is evaluated once and is not re-assignable.

The syntax for assigning a value to a named field is:

```
/FIELD=(string=fieldname)
/FIELD=(string=expression)
/FIELD=(animal="dog")
/FIELD=(amount=123)
```

13 String-level Obfuscation

To support more complex data transformation requirements, **FieldShield** supports the following field-level functions and options at the `/INREC` or output level of a job script.

ASCII Substrings
Identify substrings of an ASCII field, and reposition and recast them.

Date Intervals
Calculate the number of days between two dates.

13.1 ASCII Substrings

You can identify substrings of an ASCII field, and reposition and recast them as required on output (or in `/INREC`).

The syntax for using substrings is as follows:

```
/FIELD=(sub_string(field_name or "string",value1,value2),POSITION=...)
```

where:

- `field_name` is a field that was specified on input, where the substring is derived from its field contents
- `"string"` is an ASCII string from which the substring will be taken
- `value1` is the offset. It can be a positive value (the default) to indicate the number of characters from the left of the field (or string) where you want your substring to begin. Or, use a negative value to indicate the number of characters from the right of the field (or string).
- `value2` is the substring length. It is the number of bytes/characters you want to include in the string once your starting point has been determined using `value1`.

NOTE

`value1` and `value2` can be field names provided as shown in the third substring in the following example.

13.1.1 INSTR

Use `INSTR` to return a value that is the position of a substring within a field. If the indicated substring is not found, `INSTR` returns 0.

The INSTR procedure has two, three, or four parameters and follows the Oracle/PLSQL guidelines. For more information, see `http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions068.htm`

The syntax for using INSTR is:

```
INSTR( string1, string2 [, start [, occurrence] ] )
```

where:

- `string1` is the string to search.
- `string2` is the substring to be found in `string1`.
- `start` is the position in `string1` where the search starts. If a `start` parameter is not given, `start` defaults to 1. If `start` is negative, INSTR searches `string1` from right to left.
- `occurrence` is the number of occurrences of substring. If omitted, occurrence defaults to 1.

The parameters can be field names or literals. The values of the fields are extracted from the data.

13.2 Date Intervals

A date interval is the number of days between any two supported date data types. This interval is obtained by subtracting 2 dates (date1 - date2). In addition, a new date value can be calculated by incrementing or decrementing a date by a whole number (date + 10).

Example 24 Calculating Date Intervals

For example, consider these date pairs as input:

```
1999-12-31,2000-12-31
2004-07-01,2004-01-01
1997-06-14,2004-06-14
2004-05-21,2003-02-28
2004-02-28,2001-02-28
1905-11-29,1992-05-31
1987-10-29,1911-09-10
```

The following script, **calculate_interval.fcl**, calculates the interval between the date pairs shown above, and also derives a new date value:

```
/INFILE=dates
    /FIELD=(date1, POSITION=1, SEPARATOR=",", ISO_DATE)
    /FIELD=(date2, POSITION=2, SEPARATOR=",", ISO_DATE)
/REPORT
/OUTFILE=date_out
    /FIELD=(date1, POSITION=1, SEPARATOR=",", ISO_DATE)
    /FIELD=(date2, POSITION=2, SEPARATOR=",", ISO_DATE)
    /FIELD=(diff_days=abs(date1 - date2), POSITION=3, SEPARATOR=",")
                    # interval
    /FIELD=(new_date=date1 + 10, POSITION=4, SEPARATOR=",", ISO_DATE)
                    # new date
```

This produces the following:

```
1999-12-31,2000-12-31,366,2000-01-10
2004-07-01,2004-01-01,182,2004-07-11
1997-06-14,2004-06-14,2557,1997-06-24
2004-05-21,2003-02-28,448,2004-05-31
2004-02-28,2001-02-28,1095,2004-03-09
1905-11-29,1992-05-31,31595,1905-12-09
1987-10-29,1911-09-10,27808,1987-11-08
```

As shown above, absolute value (abs) was used to return a positive integer as the date interval, regardless of whether the earlier date in the pair appears first in the input record. The final field reflects the addition of 10 days added to the `date1` field.

14 Mathematical Data Obfuscation

In the output section of a job script, a mathematical expression can be used in place of the field name, or within a `/DATA` statement. These expressions can reference multiple fields and use arithmetic operators and mathematical functions. Use parentheses to control operator precedence, and use temporary fields to hold intermediate values. These features are particularly useful for obfuscating numeric field values using custom formulae, as illustrated in *Example 31* on page 107.

The two syntax options when using mathematical expressions are:

Introducing a new field name to store the expression

You can create a new `/FIELD` name (a field name that does not exist on input), and define its value using a mathematical expression. For example, you can use:

```
/FIELD=(T=a * 50,POSITION=1,SIZE=4.2,NUMERIC)
```

This is useful if you intend to store the derived value for use in a subsequent operation (as shown using field T in *Example 25* **on page 85**).

Using an expression in place of a field name

Rather than specifying an existing field name, you can specify a mathematical formula in a `/FIELD` or `/DATA` statement.
For example, you can use:

```
/FIELD=(a * 50,POSITION=1,SIZE=4.2,NUMERIC)
```
or
```
/DATA=a * 50
```
(as shown in the final output field in *Example 25* on page 85).

NOTE

All arithmetic symbols must be preceded and followed by a space in `/FIELD` and `/DATA` expressions, as shown in *Example 25* on page 85.

You can use the `/ROUNDING` statement to change the way that numeric values with several decimal places are rounded after an arithmetic **FieldShield** operation (see *on page 86*).

Table 5 shows the arithmetic operators in high-to-low-precedence order:

Table 5: Arithmetic Symbols and their Precedence

Opera-tor	Meaning
( )	parentheses
* / %	multiply, divide, whole number remainder of x/y
+ -	add, subtract, and unary operators (such as t=-5)

Example 25 Using Mathematical Expressions

Consider the input file, **values.in**:

```
1  2  3  4
2  3  4  5
3  4  5  6
4  5  6  7
```

The following script, **expr.fcl**, demonstrates expression writing and its use of precedence:

```
/INFILE=values.in
    /FIELD=(A, POSITION=01, SIZE=3)
    /FIELD=(B, POSITION=04, SIZE=3)
    /FIELD=(C, POSITION=07, SIZE=3)
    /FIELD=(D, POSITION=10, SIZE=3)
/REPORT
/OUTFILE=values.out
    /FIELD=(A, POSITION=01, SIZE=3)
    /FIELD=(B, POSITION=04, SIZE=3)
    /FIELD=(C, POSITION=07, SIZE=3)
    /FIELD=(D, POSITION=10, SIZE=3)
    /FIELD=(T=A + B *(C + D), POSITION=13, SIZE=9, PRECISION=0, TYPE=NUMERIC)
                 # calculate t
    /FIELD=(T2=(T - 1) / 4, POSITION=22, SIZE=12, TYPE=NUMERIC)
                 # calculate and display
```

Internal arithmetic is performed in floating-point precision. This example calculates a value for field T and displays it. The arithmetic that produces field t is performed in the following order:

1) Add the value of field c to field d.
2) Multiply by the value of field b.
3) Add the value of field a.
4) Store the value T.

This produces **values.out**:

```
1   2   3   4         15         3.50
2   3   4   5         29         7.00
3   4   5   6         47        11.50
4   5   6   7         69        17.00
```

Note that:

- The fifth column (values 15, 29, 47, and 69) was assigned the new field name T (using T= in the `/FIELD` statement to introduce the new name), and uses a complex mathematical expression involving existing field names. See above for the order of precedence used in calculating the expression.
- The final column represents a new value that was calculated using an expression involving the newly derived value of T.

The functions supported by **FieldShield** are shown in *Table 6*.

Table 6: Mathematical Functions

Function	Description
`abs(x)`	absolute value of *x*, |*x*|. *x* can be a whole number or a floating point value
`ceil(x)`	smallest whole number not less than *x*
`floor(x)`	largest whole number (as a double-precision number) not greater than *x*
`length(n)`	Return the field (*n*) length.
`mod(x,y)`	remainder of the division of *x* by *y*: *x* if *y* is zero or if *x*/*y* would overflow; otherwise the number *f* with the same sign as *x*, such that *x*=*iy*+*f* for a whole number *i*, and *f* < |*y*|
`pow(x,y)`	*x* to the power of *y*; if *x*<0, *y* must be a whole number

Rounding

Different hardware systems produce different results as a result of rounding arithmetic operations. These differences occur in the 12th or 13th decimal digit, but can become apparent after truncation.

The `/ROUNDING` statement, which should be placed at or near the top of your **FieldShield** script, is used to control how rounding is to be performed throughout the execution of the job. One of two options are available:

`/ROUNDING=SYSTEM`	The default. This produces results in the manner determined by your operating system.

`/ROUNDING=NEAREST`	For display/output purposes (only) for the results of an arithmetic operation, this option causes half a decimal digit to be added to the least significant digit. For example, if an actual result is .794999, the `NEAREST` option will cause a display of .80 when precision is specified as 2.

NOTE

The `ROUNDING=NEAREST` option slows numeric output. If your calculations do not require this level of accuracy, or if your system rounds in your preferred manner by default, then `/ROUNDING=NEAREST` should not be specified.

15 Custom Masking Functions

In addition to the supplied encryption and decryption routines (see ASCII/ALPHANUM Encryption and Decryption *on page 38*), **FieldShield** allows you to invoke your own custom field-level function that can be loaded into **FieldShield** at runtime from a **.dll** (Windows) or a **.so** (UNIX or Linux). This allows you to protect field values in ways that are not natively supported by **FieldShield**.

NOTE

For complete details on writing a custom procedure that can be invoked by **FieldShield**, see Writing Libraries For Custom Data Protection *on page 88*.

To ensure that the correct libraries are loaded, copy the appropriate library files from **/lib** in the home directory, to **/lib/modules**. If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The file can also be referenced from the current working directory.

Field-level routines must be invoked in the output section of a script. The syntax is as follows:

```
/FIELD=(field2=custom_function(arg1,arg2,arg3,...))
```

where

`field2` is the name of the derived field to contain the custom-manipulated values. *`custom_function`* is the name of a user-written function in the user-specified **.dll** or **.so**, which sets up the arguments to be specified.

`custom_function` can have any number of arguments. An argument can be any of the following:

- another **FieldShield** field
- a literal string or a constant number
- a **FieldShield**-supported function or another custom function
- an arithmetic operation whose operands can be a field, a literal, or a function

The following are examples of valid custom field function specifications:

```
• /FIELD=(field2=[handle]:custom_function(fieldx,fieldy,120,"xyz")
• /FIELD=(field2=[handle]:custom_function1   \
        (fieldx,custom_function2(fieldy,"abc"))
• /FIELD=(field2=[handle]:custom_function1    \
        (fieldx + custom_function2("xyz"))
• /FIELD=(field2=[handle]:custom_function1(custom_function2  \
        (custom_function3(sub_string("xyz",1,3),fieldx)))
```

15.1 Writing Libraries For Custom Data Protection

FieldShield allows you to invoke custom field-level functions that can be loaded at runtime via a dynamic linked library on Windows (**.dll**), or a shared object or shared library on UNIX or Linux (**.so** or **.sl**). This allows you to modify field values in ways that are not natively supported by **FieldShield**. And because the custom function is at the field level, once initialized, it is called for every field -- for every record -- that exists in the input stream.

15.2 Syntax

To ensure that the correct libraries are loaded by **FieldShield** at runtime, copy the appropriate library from **/lib** in the home directory, to **/lib/modules**.

For details on calling a custom function from a **FieldShield** job script, Custom Masking Functions *on page 87*.

The following sections describe the implementation process, in the order it is performed:

1) Custom Procedure Declaration *on page 89*.
2) Custom Procedure Arguments *on page 89*.
3) Sequence of Custom Procedure Calls *on page 94*.

15.3 Custom Procedure Declaration

The procedure declaration for a custom routine is as follows:

```
int procedure_name( cs_ext_source_t *source, cs_ext_result_t *result );
```

For Windows users, the function needs to be exported either in a **.def** file, or by using `__declspec (dllexport)`.

The first argument is a pointer to the structure `cs_ext_source_t`, which contains the source arguments. The source arguments are passed into the custom procedure using the **FieldShield** script (see Custom Masking Functions *on page 87*). These are then internally set up in the `cs_ext_source_t` structure to be passed to the custom procedure.

The second argument is a pointer to the structure `cs_ext_result_t`, which holds the result of the field-level transformation. This needs to be set up by the custom procedure after performing the transformation, and then passed back to **FieldShield**.
The `cs_ext_result_t` structure also makes provisions for sending back error codes and error messages.

The return value is an integer. The return codes are defined in the header file **sortcl_routine.h**.

The two structures `cs_ext_source_t` and `cs_ext_result_t`, and their elements, are explained in further detail below, and their declarations can be found in the header file **sortcl_routine.h**

15.4 Custom Procedure Arguments

This section describes the following:

- Source Arguments
- Result Argument *on page 92*
- Return Value *on page 93*

15.4.1 Source Arguments

The source arguments are of the type `cs_ext_source_t`:

Table 7: Source Arguments Structure

```
typedef struct cs_ext_source_s {
  void*          pPrivate;
  cs_ext_mode    iMode;
  int            iNumArgs;
  cs_ext_arg_t*  SrcArgs;
} cs_ext_source_t;
```

pPrivate

`pPrivate` is a void pointer that can be used by the custom procedure to store its own structures or data, so that the values can be retained within function calls.

iMode

`iMode` can have the following values, as also indicated in the header file:

Table 8: List of Possible Modes

```
 typedef enum {
   CS_EXT_BEGIN, /* a signal to initialize, before any data is sent
*/
   CS_EXT_VALUE,   /* used on every call to return a field or value
*/
   CS_EXT_BREAK,  /* used on break conditions, not implemented yet
*/
   CS_EXT_END     /* used to signal the end of processing, clean up
*/
 } cs_ext_mode;
```

`CS_EXT_BEGIN` indicates an initialization call to the custom procedure.

`CS_EXT_VALUE` indicates that **FieldShield** is now passing in the function arguments to the custom procedure. If any of the arguments are field names, arithmetic operations or other function calls, they are resolved before this, so that the literal values can be passed in.

`CS_EXT_END` is a finish call to the custom procedure, indicating that the records have now been processed.

`CS_EXT_BREAK` is reserved for future use.

The `iMode` values must not be modified by the custom procedure.

iNumArgs

`iNumArgs` is the total number of arguments passed to the custom procedure, for example:

```
/FIELD= (encode_hex(field1,":"))
```

In this case, the custom procedure `encodeHex` is being called with two arguments,
and therefore `iNumArgs` will be set to 2 (when the `iMode` is `CS_EXT_VALUE`) to indicate the number to arguments passed.

SrcArgs

`SrcArgs` is an array that holds the source arguments and their properties. These are set up and passed into the custom procedure from **FieldShield**, for every record, when `iMode` is `CS_EXT_VALUE`.

The structure containing the source arguments has the following elements:

Table 9: External Arguments Structure

```
/* This contains the definitions associated with the data that is
passed into the external function. */
typedef struct cs_ext_arg_s {
  int             iValueForm;
  int             iValueType;
  int             iValueLen;
  cs_ext_value_t* arg;
} cs_ext_arg_t;
```

The first two elements, `iValueForm` and `iValueType`, point to the form and the data type of the argument being passed in (see **cosort.h** for a list of all possible values).

The element `iValueLen` holds the length of the argument being passed in.

The final element, `cs_ext_value_t* arg`, holds the literal value. Depending on the data type, **FieldShield** populates the correct element of the `cs_ext_value_t` union:

Table 10: External Arguments Value Structure

```
typedef union cs_ext_value_s {
  char            *szValue;
  cs_longdouble_t ldValue;
  double          dValue;
  int             iValue;
  long            lValue;
  short           shValue;
} cs_ext_value_t;
```

15.4.2 Result Argument

The result arguments are of the type `cs_ext_result_t`:

Table 11: Result Argument Structure

```
typedef struct cs_ext_result_s {
  int          iErrorNum;
  char         *szErrorMsg;
  cs_ext_arg_t *ResArg;
} cs_ext_result_t;
```

ResArg

The `ResArg` element, also of the type `cs_ext_arg_t`, stores the result value that needs to passed back to **FieldShield** in the event of successful processing.

The structure containing the result argument has the following elements:

Table 12: External Arguments Structure

```
/* This contains the definitions associated with the data that is
passed into the external function. */
typedef struct cs_ext_arg_s {
  int             iValueForm;
  int             iValueType;
  int             iValueLen;
  cs_ext_value_t* arg;
} cs_ext_arg_t;
```

The first two elements, `iValueForm` and `iValueType`, point to the form and the data type of the argument being passed back to **FieldShield** (see **cosort.h** for a list of all possible values).

The element `iValueLen` holds the length of the argument being passed back.

The final element, `cs_ext_value_t* arg`, holds the actual result value. Depending on the return data type, the custom procedure needs to insert the value into the correct element of the `cs_ext_value_t` union, and then return it to **FieldShield**:

Table 13: External Arguments Value Structure

```
typedef union cs_ext_value_s {
  char            *szValue;
  cs_longdouble_t ldValue;
  double          dValue;
  int             iValue;
  long            lValue;
  short           shValue;
} cs_ext_value_t;
```

Error Handling

The first two elements of the result structure, `iErrorNum` and `szErrorMsg`, can be used to pass back an error code and an error message string to be displayed by **FieldShield**. `iErrorNum` can be any integer, and the element `szErrorMsg` must point to a null-terminated string.

15.4.3 Return Value

The following table lists the return values expected by **FieldShield**:

Table 14: Return Values

```
#define CE_EXT_OK          0
#define CE_EXT_BADTYPE     1
#define CE_EXT_BADVALUE    2
#define CE_EXT_BADNUM      3
#define CE_EXT_ABORT      -1
```

The return value `CE_EXT_OK` indicates success.

`CE_EXT_BADTYPE` indicates that the data type of an argument being passed was incorrect. In this case, a warning is generated, and any further calls for the remaining records to the custom procedure are bypassed.

`CE_EXT_BADVALUE` and `CE_EXT_BADNUM` indicate an incorrect string or numeric value, and both generate warning messages for that particular record. No further calls
to the custom procedure are bypassed.

On receiving a return value of `CE_EXT_ABORT` from the custom procedure, **FieldShield** will cleanup its resources and exit the program.

15.5 Sequence of Custom Procedure Calls

FieldShield makes three primary sets of calls to the custom routine in the following order:

1) Initialization Call.
2) Processing field-level transformations for each record.
3) Termination *on page 95*.

15.5.1 Initialization Call

At first, the custom function is called with `iMode` set to `CS_EXT_BEGIN` (see *Table 8* on page 90 for a list of all the modes).

At this point, the custom procedure can set up its own resources and initialization parameters. It can then set the `pPrivate` pointer to point to any allocated resources, so that it is passed in, from this point on, for all the records that will be processed. The `pPrivate` pointer is for use with the custom procedure only and is not altered by **FieldShield**.

15.5.2 Processing field-level transformations for each record

Once the initialization is successful, **FieldShield** will set up the source arguments structure for each record, and call the custom procedure with `iMode` set to `CS_EXT_VALUE`.

At this point, **FieldShield** has populated the source arguments structure (*Table 7* on page 90) with all the necessary values.

For example, assume that the source argument variable is named `cs_source`.

`cs_source->pPrivate` is the pointer set up by the custom procedure during the initialization call.

The element `cs_source->iNumArgs` contains the total number of arguments that are being passed in.

These arguments are stored in the array `cs_source->SrcArgs`, and can be accessed using the array index. For example, the first argument passed can be accessed using `cs_source->SrcArgs[0]`, and so on.

Each source argument has its form, data type, length and actual value passed in.

For example, the length of the first argument can be accessed using `cs_source->SrcArgs[0].iValueLen`.

The actual argument values are stored in the argument structure (see *Table 13* on page 93) for that particular source element. For example, if the first source argument is a string, its value can be obtained from `cs_source->SrcArgs[0].arg->szValue`.

Once the custom procedure makes sure it has the correct arguments, it can finish the processing and set up the result argument (see *Table 14* on page 93) to pass back the protected values to **FieldShield**.

For example, assume here that the result argument is named `cs_result`.

The final result argument value is stored in `cs_result->ResArg`. The custom procedure needs to set up the form, data type, and length in order for the result to be passed back to **FieldShield**.

The actual computed value is stored in `cs_result->ResArg->arg`. For example, if the custom procedure is passing back a string result to **FieldShield**, the final value should be stored in `cs_result->ResArg->arg->szValue`.

15.5.3 Termination

After all the records have been sent for processing to the custom procedure, **FieldShield** makes a final call to the custom procedure with `iMode` set to `CS_EXT_END`
(see *Table 8* on page 90 for a list of all the modes).

At this point, the custom procedure should free any allocated resources. If you are executing with `THREADMAX` > 1, then this call is made once for each thread.

This procedure is illustrated in Example *on page 95*.

15.5.4 Example

This example is divided into two parts. The first is the **FieldShield** script containing the call to the custom procedure. The second is the C program that explains how the custom procedure is implemented.

FieldShield Script Using a Custom Field-Level Function

```
/INFILE=data.in
    /FIELD=(raw_data,POSITION=1,SIZE=10)
/REPORT
/OUTFILE=data.out
    /FIELD=(encoded_data=encodeHex(raw_data,":"),POSI-
TION=1,SIZE=30)
```

In the output file, the derived field encoded_data is generated by calling the function `encodeHex` with two arguments: the first is the input field raw_data, and the second is the literal string `":"`.

The following code is an implementation of the custom routine `encodeHex` in C, which can be compiled into **ext_func.dll** on Windows and **ext_func.so** (**.sl**) on UNIX.

Note the function:

```
_LIBSPEC int encodeHex( cs_ext_source_t *source, cs_ext_result_t *result )
```

in order to understand how arguments are resolved and used to perform the protection, where results sent back to the **FieldShield** program.

C Program to Perform a Custom Procedure

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * ext_func.c - example plugin for field level transformations
 *
 * Copyright (c) 2008, Innovative Routines International, Inc.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Defines an external function that encodes a possibly
 * binary field into ASCII hex, replacing every byte with two
 * characters 0-f.
 *
 * Also demonstrates using optional parameters in an external
 * function. If called with a character string as the second
 * parameter, the first character of the string will be used
 * as a separator.
* * * * * * * * * * * * * * * * * * * *  * * * * * * * * * * * /
/* Define macros to allow exported functions under both
   Windows or UNIX
 */
#if defined (_WIN32)
#define _LIBSPEC __declspec (dllexport)
#include <windows.h>
#else
#define _LIBSPEC
#endif /* _WIN32 */

#include <memory.h>

#include "cosort.h"
#include "sortcl_routine.h"
/*
 * Handle field lengths up to 256 bytes.
 * Encoding replaces every byte with two characters
 * (or three with optional separator)
 * so we need up to three times the space for transformed fields.
 */
#define MAX_FIELD_IN 256
#define MAX_FIELD_OUT (MAX_FIELD_IN * 3)

static const char chex[] = "0123456789abcdef";

/*
 * This helper function is called locally and need not be
 * exported. Returns the number of characters in the encoded
 * string.
 */
int hex_encode( unsigned char *inb, char *outb, const int len, const char
sep )
{
```

continued on next page...

```
       int n, iI, iO;

       iO = 0;
       iI = 0;

while (iI < len) {
              n = ((inb[iI] >> 4) & 0x0f);
              outb[iO++] = chex[n];
              n = (inb[iI++] & 0x0f);
              outb[iO++] = chex[n];
              /* do not put separator at end of string */
              if (('\0' != sep) && (iI < len)) {
                  outb[iO++] = sep;
              }
       }
       return (iO);
}

/*
 * This is the exported function that will be referenced in
 * the FieldShield FIELD statement. It receives a field value in
 * the source structure and returns the hex encoded field in
 * the result structure.
 */
_LIBSPEC int encodeHex( cs_ext_source_t *source,
                        cs_ext_result_t *result )
{
  int n;
  int iRet = CE_EXT_OK;
  char *p;
  char c = '\0';

  if (CS_EXT_BEGIN == source->iMode) {
    /* first call to external function, allocate
       memory for field out */
    p = (char *)malloc( MAX_FIELD_OUT );
    if (NULL == p) {
      iRet = CE_EXT_ABORT;
    }
    else {
      source->pPrivate = p;
    }
  }
  else if (CS_EXT_END == source->iMode) {
    /* last call to function, clean up */
    free( source->pPrivate );
    source->pPrivate = NULL;
  }
```

continued on next page...

```
 else {
    p = source->pPrivate;
    if (NULL != p) {
      if (1 < source->iNumArgs) {
        c = source->SrcArgs[1].arg->szValue[0];
       }
       n = hex_encode( (unsigned char *)
                         source->SrcArgs[0].arg->szValue,
                   p,
                   source->SrcArgs[0].iValueLen,
                   (const)c );

      result->ResArg[0].iValueLen = n;
      result->ResArg[0].iValueType = CS_ASCII;
      result->ResArg[0].arg->szValue = p;
    }
    else {
      /* we have lost our private data pointer... abort! */
      iRet = CE_EXT_ABORT;
    }
  }
  return (iRet);
}
```

16 FieldShield Examples

Examples in this chapter include:

- Remapping *on page 102*
- Creating Multiple Output Files *on page 103*
- Redaction *on page 104*
- De-identifying Data *on page 105*
- Re-identifying Data *on page 106*
- Anonymization Using Mathematical Obfuscation *on page 107*
- Pseudonymization *on page 108*
- Restoring Pseudonymized Values *on page 109*
- Field Masking *on page 111*
- Conditional Field Masking *on page 112*
- Encrypting Individual Fields *on page 113*
- Decrypting Individual Fields *on page 114*
- Encrypting an Entire Record *on page 115*
- Decrypting an Entire Record *on page 116*
- Conditional Field Encryption *on page 117*
- Multiple Field Protections in One Pass *on page 118*
- GPG Encryption *on page 119*
- GPG decryption *on page 120*

These examples refer to the input file **seqdata.in**, which contains:

```
01 775-17-0363 Stuart Clay       56681.42 6 CT 101 B St
02 810-90-1269 Taylor Guerrero   15019.10 9 MD 1031 Park Ln Apt D
03 906-43-3545 Charles Caldwell  41116.71 3 NY 14 Main St
04 787-73-7773 Robyn Puckett     44558.62 3 NY 822 Hwy 76
05 950-52-1240 Santiago Lindsey  55836.11 0 TX Star Rt Box 822
06 891-56-9799 Charles Lindsey   98525.58 2 TX 12746 Wolf Circle
07 789-12-7387 Santiago Puckett  59036.80 4 NY 321 Baltic Ct
08 899-60-4065 Charles Williams  18645.95 1 TX 1103 Fresh Creek Ln
09 951-05-4521 Jack Velazquez    29205.44 2 NY 6780 Sand Dr Apt 3A
10 788-35-4977 Donald Cooke      44121.44 4 MA 35 La Palma Dr
```

The following metadata file, **seqdata.ddf**, describes the above input file (see Data Definition Files *on page 143*):

```
/FIELD=(idnum,POSITION=1,SIZE=2.0,NUMERIC) # Unique record identifier tag
/FIELD=(ssno,POSITION=4,SIZE=11)
        # Breaks the social security number into parts for obfuscation:
/FIELD=(ssno_part1,POSITION=4,SIZE=3)
/FIELD=(ssno_part2,POSITION=8,SIZE=2,NUMERIC)
/FIELD=(ssno_part3,POSITION=11,SIZE=4,NUMERIC)
/FIELD=(name,POSITION=16,SIZE=18)
/FIELD=(salary,POSITION=34,SIZE=8,NUMERIC)
/FIELD=(salary2,POSITION=34,SIZE=8)  # Re-defined as ASCII for encryption
/FIELD=(deduction_no,POSITION=43,SIZE=1)
/FIELD=(state,POSITION=45,SIZE=2)
/FIELD=(address,POSITION=48,SIZE=20)
/FIELD=(fieldgroup,POSITION=1,SIZE=42)
        # Define the first four fields together
```

Example 26 Remapping

The following script, **remapping.fcl**, converts the fixed-position layout of the file **seqdata.in** to a pipe-separated layout, rearranges the fields, and leaves out unwanted fields:

```
/INFILE=seqdata.in
    /SPEC=seqdata.ddf # read in fixed-position metadata from separate .DDF
/REPORT
/OUTFILE=seqdata_remapped.out
    /FIELD=(SALARY, POSITION=1, SEPARATOR="|", NUMERIC)
    /FIELD=(NAME, POSITION=2, SEPARATOR="|", RIGHT_ALIGN)
    /FIELD=(SSNO, POSITION=3, SEPARATOR="|")
```

The file **seqdata_remapped.out** is as follows:

```
56681.42|Stuart Clay|775-17-0363
15019.10|Taylor Guerrero|810-90-1269
41116.71|Charles Caldwell|906-43-3545
44558.62|Robyn Puckett|787-73-7773
55836.11|Santiago Lindsey|950-52-1240
98525.58|Charles Lindsey|891-56-9799
59036.80|Santiago Puckett|789-12-7387
18645.95|Charles Williams|899-60-4065
29205.44|Jack Velazquez|951-05-4521
44121.44|Donald Cooke|788-35-4977
```

Note that:

- The output fields are pipe-separated. This was achieved by using the SEPARATOR attribute in the output section, along with the POSITION attribute to specify the field numbers (see POSITION *on page 148* and SEPARATOR *on page 150*). Note that the SIZE attribute is not required when defining delimited fields.
- The name field has been trimmed of trailing white space with the `RIGHT_ALIGN` attribute (see Alignment *on page 157*).
- The output fields are arranged in a different order than found on input, for example, with salary occurring first. The POSITION attribute is used to place fields in the desired order on output.
- Several fields from the input file were omitted from the output description, and are therefore not contained in the output file.

Example 27 Creating Multiple Output Files

The following script, **multiple_out.fcl**, produces two output files in a single execution: one will contain fields with a set of information that includes social security numbers. The second output file will contain fields with other information, including addresses:

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/REPORT
/OUTFILE=user_id.out
    /FIELD=(IDNUM, POSITION=1, SIZE=2, SEPARATOR="\t")
    /FIELD=(SSNO, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t", RIGHT_ALIGN)
/OUTFILE=user_address.out
    /FIELD=(NAME, POSITION=1, SEPARATOR="\t", RIGHT_ALIGN)
    /FIELD=(SALARY, POSITION=2, SEPARATOR="\t")
    /FIELD=(ADDRESS, POSITION=3, SEPARATOR="\t")
    /FIELD=(STATE, POSITION=4, SEPARATOR="\t")
```

The output file **user_id.out** contains:

```
01 775-17-0363 Stuart Clay
02 810-90-1269 Taylor Guerrero
03 906-43-3545 Charles Caldwell
04 787-73-7773 Robyn Puckett
05 950-52-1240 Santiago Lindsey
06 891-56-9799 Charles Lindsey
07 789-12-7387 Santiago Puckett
08 899-60-4065 Charles Williams
09 951-05-4521 Jack Velazquez
10 788-35-4977 Donald Cooke
```

The output file **user_address** contains:

```
Stuart Clay         56681.42  101 B St            CT
Taylor Guerrero     15019.10  1031 Park Ln Apt D  MD
Charles Caldwell    41116.71  14 Main St          NY
Robyn Puckett       44558.62  822 Hwy 76          NY
Santiago Lindsey    55836.11  Star Rt Box 822     TX
Charles Lindsey     98525.58  12746 Wolf Circle   TX
Santiago Puckett    59036.80  321 Baltic Ct       NY
Charles Williams    18645.95  1103 Fresh Creek Ln TX
Jack Velazquez      29205.44  6780 Sand Dr Apt 3A NY
Donald Cooke        44121.44  35 La Palma Dr      MA
```

Note that:

- Two `/OUTFILE` sections were specified to produce two distinct output files, each with its own set of `/FIELD` statements (see Output Files *on page 142*).
- The fields in **user_addresses.out** were remapped using POSITION attributes that were different on output than on input (see POSITION *on page 148*).

Example 28 Redaction

This following script, **redact.fcl**, demonstrates how you can redact data by omitting one or more fields from the output file description:

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf
  # read in metadata from separate .DDF
/REPORT
/OUTFILE=redact_sensitive.out
    /FIELD=(idnum,POSITION=1,SIZE=2.0,NUMERIC)
    /FIELD=(name,POSITION=4,SIZE=18)
  # Fields repositioned from source
    /FIELD=(state,POSITION=23,SIZE=2)
    /FIELD=(address,POSITION=26,SIZE=20)
```

This produces the output file, **redact_sensitive.out**:

```
01 Stuart Clay        CT 101 B St
02 Taylor Guerrero    MD 1031 Park Ln Apt D
03 Charles Caldwell   NY 14 Main St
04 Robyn Puckett      NY 822 Hwy 76
05 Santiago Lindsey   TX Star Rt Box 822
06 Charles Lindsey    TX 12746 Wolf Circle
07 Santiago Puckett   NY 321 Baltic Ct
08 Charles Williams   TX 1103 Fresh Creek Ln
09 Jack Velazquez     NY 6780 Sand Dr Apt 3A
10 Donald Cooke       MA 35 La Palma Dr
```

Note that:

- Several of the fields present in the input file, including social security number and salary, have been omitted by failing to include the unwanted `/FIELD` statements in the output file description.
- The fields that were included in the output file description have been repositioned to accommodate for the fields that have been omitted (see POSITION *on page 148*).

Example 29 De-identifying Data

In this example, **deid.fcl**, the name field from **seqdata.in** will be de-identified. Note that this provides a low-security protection layer (for maximum field protection, see Encrypting Individual Fields *on page 113* and Encrypting an Entire Record *on page 115*):

```
/INFILE=seqdata.in
   /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/REPORT
/OUTFILE=namesdeid.out
    /FIELD=(IDNUM,POSITION=1, SEPARATOR="\t")
    /FIELD=(SSNO,POSITION=2, SEPARATOR="\t")
    /FIELD=(ID_NAME=de_identify(NAME, "12345678"), POSITION=3, \
            SEPARATOR="\t") # key is 12345678
```

This produces the file **namesdeid.out**:

```
01 775-17-0363 OFU*nFK?>*BKKKKKKK
02 810-90-1269 J*B>DnKQUqnnqnDKKK
03 906-43-3545 ?d*n>q;K?*>6Lq>>KK
04 787-73-7773 |D^B}KIU+3qFFKKKKK
05 950-52-1240 O*}F2*<DKH2}6;qBKK
06 891-56-9799 ?d*n>q;KH2}6;qBKKK
07 789-12-7387 O*}F2*<DKIU+3qFFKK
08 899-60-4065 ?d*n>q;K,2>>2*y;KK
09 951-05-4521 x*+3K~q>*v:UqvKKKK
10 788-35-4977 @D}*>6K?DD3qKKKKKK
```

Note that:

- the name field in the final column has been de-identified (see Randomization *on page 72*). It is unreadable and the size may be variable due to non-printable characters in the de-identified data.
- the key used to de-identify the name field was `"12345678"`. This key must be used in any subsequent re-identification job (as in *Example 30* on page 106).

Example 30 Re-identifying Data

The following script, **reid.fcl**, reads the **namesdeid.out** file (created in *Example 29* on page 105), and returns the original name field values on output:

```
/INFILE=namesdeid.out # file with de-identified field data
    /FIELD=(IDNUM, POSITION=1, SEPARATOR="\t")
    /FIELD=(SSNO, POSITION=2, SEPARATOR="\t")
    /FIELD=(ID_NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE=namesreid.out # output file with re-identified data
    /FIELD=(IDNUM, POSITION=1, SEPARATOR="\t")
    /FIELD=(SSNO, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME=re_identify(ID_NAME, "12345678"), POSITION=3,\
            SEPARATOR="\t", RIGHT_ALIGN)
            # same key is used: 12345678
```

This produces the following, namesreid.out:

```
01 775-17-0363 Stuart Clay
02 810-90-1269 Taylor Guerrero
03 906-43-3545 Charles Caldwell
04 787-73-7773 Robyn Puckett
05 950-52-1240 Santiago Lindsey
06 891-56-9799 Charles Lindsey
07 789-12-7387 Santiago Puckett
08 899-60-4065 Charles Williams
09 951-05-4521 Jack Velazquez
10 788-35-4977 Donald Cooke
```

The original name field is restored because it was re-identified with the original de-identification key (see Randomization *on page 72*).

Example 31 Anonymization Using Mathematical Obfuscation

The following **FieldShield** job script, **math_ob.fcl**, obfuscates the social security field in **seqdata.in** using mathematical formulae. Note that this provides a low-security protection layer (for maximum field protection, see Encrypting Individual Fields *on page 113* and Encrypting an Entire Record *on page 115*):

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/INREC
    /FIELD=(IDNUM, POSITION=1, SIZE=2) #, PRECISION=0, NUMERIC)
    /FIELD=(SSNO_PART1, POSITION=4, SIZE=3)
    /FIELD=(NEW_SSNO_PART2, POSITION=8, SIZE=2, PRECISION=0, FILL='0', \
            NUMERIC, \
        IF SSNO_PART2 GT 50 THEN SSNO_PART2 / 2 ELSE 2 * SSNO_PART2 - 5)
    /FIELD=(NEW_SSNO_PART3, POSITION=11, SIZE=4, PRECISION=0, FILL='0', \
            NUMERIC, \
        IF SSNO_PART3 GT 4500 THEN SSNO_PART3 / 2 ELSE 2 * SSNO_PART3 - 54)
   /FIELD=(NAME, POSITION=16, SIZE=18)
/REPORT
/OUTFILE=seqdata_ssobfuscate.out
    /FIELD=(IDNUM, POSITION=1, SEPARATOR="\t")
    /FIELD=(SSNO=format_strings("%s-%s-%s", SSNO_PART1, NEW_SSNO_PART2, \
            NEW_SSNO_PART3), POSITION=2, SEPARATOR="\t")
   /FIELD=(TRIM_NAME=trim(NAME), POSITION=3, SEPARATOR="\t")
```

This produces the following output file, **seqdata_ssobfuscate.out**:

```
01 775-29-0672 Stuart Clay
02 810-45-2484 Taylor Guerrero
03 906-81-7036 Charles Caldwell
04 787-37-3887 Robyn Puckett
05 950-26-2426 Santiago Lindsey
06 891-28-4900 Charles Lindsey
07 789-19-3694 Santiago Puckett
08 899-30-8076 Charles Williams
09 951-05-2261 Jack Velazquez
10 788-65-2489 Donald Cooke
```

Note that:

- The social security field is obfuscated, as seen when compared to the original values found in **seqdata.in**. Using IF THEN ELSE field-level logic, parts of the social security field (ssno_part2 and ssno_part3) were conditionally obfuscated with different formulae (see Conditional Field Security and Masking *on page 19*). For example:

  - The second part of Stuart Clay’s social security number converted from 17 in the input file to 29. Because the input value was under 50, this formula was applied: `2 * ssno_part2 - 5`.
  - The second part of Taylor Guerrero’s social security number converted from 90 in the input file to 45. Because the input value was over 50, this formula was applied: `ssno_part2 / 2`.

- `/DATA` statements were used to re-insert the necessary dashes in the social security number field, and to insert a space before the final name field (see /Re-formatting and Masking Options *on page 74*).
  Note that `POSITION` attributes are not required for the fields when formatting the output record with a sequence of `/FIELD` and `/DATA` statements.

NOTE

Due to the conditional (IF THEN ELSE) application of the different formulae, no single reverse formula can be applied to restore the obfuscated values.

Example 32 Pseudonymization

This example demonstrates how **FieldShield** SET files (look-up tables) can be used to perform pseudonymization. The file **pseudo.set** is required to perform this example:

```
Charles Caldwell        Teddy Black
Charles Lindsey         Landen Sullivan
Charles Williams        Jeffery Gomez
Donald Cooke            Julio Koch
Jack Velazquez          Francisco Duffy
Robyn Puckett           Salvador Jacobson
Santiago Lindsey        Spencer Craig
Santiago Puckett        Alvaro Mcleod
Stuart Clay             Ramsey Flynn
Taylor Guerrero         Clifton Jimenez
```

In the above SET file, the real name Charles Caldwell is matched with the pseudonym Teddy Black, and so on. Note that a tab character separates each real first and last name from its pseudonym first and last name.

The following script, **pseudo.fcl**, invokes the above SET file to replace each real name with its pseudonym:

```
/INFILE=seqdata.in
   /SPECIFICATION=seqdata.ddf  # read in metadata from separate .DDF
/REPORT
/OUTFILE=seqdata_pseudonyms.out
   /FIELD=(IDNUM, POSITION=1, SEPARATOR="\t")
           # Unique record identifier tag
   /FIELD=(SSNO, POSITION=2, SEPARATOR="\t")
   /FIELD=(NAME_FAKE, POSITION=3, SEPARATOR="\t", SET=pseudo.set[name])
           # invokes look-up table
```

This produces the output file **seqdata_pseudonynms.out**:

```
01 775-17-0363 Ramsey Flynn
02 810-90-1269 Clifton Jimenez
03 906-43-3545 Teddy Black
04 787-73-7773 Salvador Jacobson
05 950-52-1240 Spencer Craig
06 891-56-9799 Landen Sullivan
07 789-12-7387 Alvaro Mcleod
08 899-60-4065 Jeffery Gomez
09 951-05-4521 Francisco Duffy
10 788-35-4977 Julio Koch
```

The final column, name_fake, used the file **pseudo.set** to replace the original name value with its pseudonym. Stuart Clay has been replaced with Ramsey Flynn, and so on (see Universally Unique Identifier *on page 61*).

Example 33 Restoring Pseudonymized Values

This example demonstrates how FieldShield SET files (look-up tables) can be used to restore previously pseudonymized values (see *Example 32* on page 108). The file **pseudo_restore.set** is required:

```
Alvaro Mcleod     Santiago Puckett
Clifton Jimenez   Taylor Guerrero
Francisco Duffy   Jack Velazquez
Jeffery Gomez     Charles Williams
Julio Koch        Donald Cooke
Landen Sullivan   Charles Lindsey
Ramsey Flynn      Stuart Clay
Salvador Jacobson Robyn Puckett
Spencer Craig     Santiago Lindsey
Teddy Black       Charles Caldwell
```

In the above SET file, the pseudonym Teddy Black is matched with the real name Charles Caldwell, and so on. Note that a tab character separates each pseudonym from the real name.

NOTE

You can create your own reverse look-up table, such as the one used in this example, with a simple **FieldShield** script, for example:

```
/INFILE=pseudo.set
    /FIELD=(real,position=1,separator='\t')
      # \t denotes a tab separator
    /FIELD=(fake,position=2,separator='\t')
/REPORT
/OUTFILE=pseudo_restore.set
    /FIELD=(fake,position=1,separator='\t')
    /FIELD=(real,position=2,separator='\t')
```

The following script, **pseudo_restore.fcl**, invokes the above SET file to restore each pseudonym with the original name:

```
/INFILE=seqdata_pseudonyms.out
    /FIELD=(IDNUM, POSITION=1, SEPARATOR="\t")
    /FIELD=(SSNO, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME_FAKE, POSITION=3, SEPARATOR="\t") # pseudonym
/REPORT
/OUTFILE=names_restored.out
    /FIELD=(IDNUM, POSITION=1, SEPARATOR="\t")
    /FIELD=(SSNO, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t", \
           SET=pseudo_restore.set[NAME_FAKE])
                # invokes the reverse look-up table
```

This produces the output file **names_restored.out**:

```
01 775-17-0363 Stuart Clay
02 810-90-1269 Taylor Guerrero
03 906-43-3545 Charles Caldwell
04 787-73-7773 Robyn Puckett
05 950-52-1240 Santiago Lindsey
06 891-56-9799 Charles Lindsey
07 789-12-7387 Santiago Puckett
08 899-60-4065 Charles Williams
09 951-05-4521 Jack Velazquez
10 788-35-4977 Donald Cooke
```

The final column, name_fake, referenced the file **pseudo_restore.set** to restore the original name based on its pseudonym. Ramsey Flynn has been replaced with Stuart Clay, and so on (see Universally Unique Identifier *on page 61*).

Example 34 Field Masking

In the following script, **mask.fcl**, `/DATA` statements with repeated constant strings are used to replace sensitive salary values:

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/REPORT
/OUTFILE=mask_salary.out
    /FIELD=(idnum,POSITION=1,SIZE=2)
    /FIELD=(ssno,POSITION=4,SIZE=12)
    /FIELD=(name,POSITION=16,SIZE=19)
    /DATA={8}"*"      # repeated constant string for redaction
```

This produces the output file, **mask_salary.out**:

```
01 775-17-0363 Stuart Clay        ********
02 810-90-1269 Taylor Guerrero    ********
03 906-43-3545 Charles Caldwell   ********
04 787-73-7773 Robyn Puckett      ********
05 950-52-1240 Santiago Lindsey   ********
06 891-56-9799 Charles Lindsey    ********
07 789-12-7387 Santiago Puckett   ********
08 899-60-4065 Charles Williams   ********
09 951-05-4521 Jack Velazquez     ********
10 788-35-4977 Donald Cooke       ********
```

Note that:

- `/DATA` statements with a repeated constant string (*) were used to mask the salary field in the last column (see /Re-formatting and Masking Options *on page 74*).
- The output fields were re-positioned using different `POSITION` attributes than those found in the .DDF to describe the input file layout (see POSITION *on page 148*).

Example 35 Conditional Field Masking

The following script, **mask_cond.fcl**, demonstrates how you can apply redaction to one or more fields on a conditional basis. Records that do not satisfy the condition are left unchanged:

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/INREC
   /FIELD=(IDNUM, POSITION=1, SIZE=2)
   /FIELD=(SSNO, POSITION=4, SIZE=11)
   /FIELD=(NAME, POSITION=16, SIZE=18)
   /FIELD=(SALARY2, POSITION=35, SIZE=8)
   /FIELD=(NO_SALARY2=replace_chars(SALARY2, "#", 1, 2), POSITION=44, \
      SIZE=8)
/REPORT
/OUTFILE=mask_high_salary.out
   /CONDITION=(HI_SALARY, TEST=(SALARY2 GE 50000))
   /FIELD=(IDNUM, POSITION=1, SEPARATOR="\t")
   /FIELD=(SSNO, POSITION=2, SEPARATOR="\t")
   /FIELD=(COND_SALARY2, POSITION=3, SEPARATOR="\t", /
    IF HI_SALARY THEN NO_SALARY2 ELSE SALARY2)
   /FIELD=(TRIM_NAME=trim(NAME), POSITION=4, SEPARATOR="\t")
```

This produces the output file, **mask_high_salary.out**:

```
01    775-17-0363   ##681.42   Stuart Clay
02    810-90-1269  15019.10   Taylor Guerrero
03    906-43-3545  41116.71   Charles Caldwell
04    787-73-7773  44558.62   Robyn Puckett
05    950-52-1240  ##836.11   Santiago Lindsey
06    891-56-9799  ##525.58   Charles Lindsey
07    789-12-7387  ##036.80   Santiago Puckett
08    899-60-4065  18645.95   Charles Williams
09    951-05-4521  29205.44   Jack Velazquez
10    788-35-4977  44121.44   Donald Cooke
```

Note that:

- In the INREC, an additional field was created where the first two characters of the SALARY2 field were replaced with # signs.
- The output fields were changed from sized fields to delimited fields. In addition, the order of the fields was changed.
- In the output a conditional field was created where No_SALARY2 is used for the value when SALARY2 is greater than 50000. Otherwise, SALARY2 is used for the value.

Example 36 Encrypting Individual Fields

This example uses a file called **Privatekey.txt** which, for this example, is located in the directory **C:\admin\private**, and has restricted access. **Privatekey.txt** contains a single line of text, the passphrase, to be used for encryption (and subsequent decryption):

```
XYZ
```

The following script, **enc_2columns.fcl**, demonstrates how you can encrypt selected column data, while leaving other column data unchanged.

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/REPORT
/OUTFILE=encrypt_2fields.out
        # Use literal passphrase:
    /FIELD=(ENC_SSNO=enc_aes256(SSNO, "ABC"), POSITION=1, SIZE=24)
    /FIELD=(NAME, POSITION=26, SIZE=18)
# Use passphrase embedded in file residing in limited-access directory:
    /FIELD=(ENC_SALARY2=enc_aes256(SALARY2, "file:./Privatekey.txt"), \
           POSITION=45, SIZE=24)
    /FIELD=(STATE, POSITION=70, SIZE=2)
```

This produces the file, **encrypt_2fields.out**:

```
yYwvJeABDSfLBkrzEu4h/g== Stuart Clay        4EVPbBq7yCxjA7BFU4i3fw==  CT
uRg+HsH9QIY8+ARFAf6aeQ== Taylor Guerrero    N/lGXJO4xgoHdaNTK7hF8w==  MD
vz4Vkd9sLDjHaLyFlvVhFg== Charles Caldwell   6wmh4D0BdQtMljeH0qNkCw==  NY
DeJmKoMR81gRkNeH4cRWhA== Robyn Puckett      FN4j256CrZLnUOzkkFkYyw==  NY
Zd54dFVjLOv3F7Tg7ZtqsA== Santiago Lindsey   uLBKz3BUaslAwRF+Z7appA==  TX
2aPYmtTzXCHwXFO5GI1YCQ== Charles Lindsey    8TCZ/zWw0kVzK4s+XLwDXQ==  TX
yh9rEhhK3XMk0SETYvj7ZA== Santiago Puckett   c9gLlG0GeL4/4WgCxqAZXw==  NY
qWzmqFxmTTuYZl76YmKfyw== Charles Williams   DYEbc+2GTPQwRKao0hBWyg==  TX
vxyb1gaFBe0cNxLp4mOwmQ== Jack Velazquez     2nqoFGjvPDIYKVXDWb/Gog==  NY
W+Azd6wuzR16DbCA6QIV0w== Donald Cooke       BHbX+IrFn24jv6kGITEWtA==  MA
```

Note that:

- The social security number and salary fields are encrypted with 256-bit encryption, while the name and state fields remain unchanged (see ASCII/ALPHANUM Encryption and Decryption *on page 38*).
- The ssno field was encrypted using a literal passphrase (lower security). See Advanced Encryption Standard (AES) *on page 42* for available options.
- The salary field was encrypted using a passphrase found in a limited-access directory (higher security).
- The field sizes of the ssno and salary fields are larger when compared to their original sizes (see Standard Encryption Field Size Considerations *on page 43*).

- The field *salary*, which was defined as ASCII on input, was used for encryption purposes, rather than the numeric field *salary*. Encryption is supported for ASCII fields only, and therefore you may need to over-define a field as ASCII, as in this example.

Example 37 Decrypting Individual Fields

The following script, **dec_2columns.fcl** decrypts the two fields that were encrypted using the script **enc_2columns.fcl** (see *Example 36* on page 113):

```
/INFILE=encrypt_2fields.out # Output file from encrypt operation
    /FIELD=(ENC_SSNO, POSITION=1, SIZE=24)
    /FIELD=(NAME, POSITION=26, SIZE=18)
    /FIELD=(ENC_SALARY2, POSITION=45, SIZE=24)
    /FIELD=(STATE, POSITION=70, SIZE=2)
/REPORT
/OUTFILE=restore_2fields.out
        # Use literal passphrase:
    /FIELD=(SSNO=dec_aes256(ENC_SSNO, "ABC"), POSITION=1, SIZE=11)
    /FIELD=(NAME, POSITION=13, SIZE=18)
# Use passphrase embedded in a file residing in limited-access directory:
    /FIELD=(SALARY2=dec_aes256(ENC_SALARY2, "file:./Privatekey.txt"), \
           POSITION=32, SIZE=8)
    /FIELD=(STATE, POSITION=41, SIZE=2)
```

This produces **restore_2fields.out**:

```
775-17-0363 Stuart Clay        56681.42 CT
810-90-1269 Taylor Guerrero    15019.10 MD
906-43-3545 Charles Caldwell   41116.71 NY
787-73-7773 Robyn Puckett      44558.62 NY
950-52-1240 Santiago Lindsey   55836.11 TX
891-56-9799 Charles Lindsey    98525.58 TX
789-12-7387 Santiago Puckett   59036.80 NY
899-60-4065 Charles Williams   18645.95 TX
951-05-4521 Jack Velazquez     29205.44 NY
788-35-4977 Donald Cooke       44121.44 MA
```

Note that:

- The social security number and salary fields display the original, restored values (see ASCII/ALPHANUM Encryption and Decryption *on page 38*).
- The literal passphrase `"ABC"` was used to decrypt the social security number because the same passphrase must be used for both encryption and decryption.

- A reference to the file **Privatekey.txt** was used to decrypt the salary field because it holds the same passphrase that was used for encryption.
- The original sizes of the ssno and salary fields were specified in the output section, while the larger encrypted field sizes were specified in the input section (see Standard Encryption Field Size Considerations *on page 43*).

Example 38 Encrypting an Entire Record

This example uses the variable-length input file **seqdata_reduced.in**:

```
01 775-17-0363 Stuart Clay
02 810-90-1269 Taylor Guerrero
03 906-43-3545 Charles Caldwell
04 787-73-7773 Robyn Puckett
05 950-52-1240 Santiago Lindsey
06 891-56-9799 Charles Lindsey
07 789-12-7387 Santiago Puckett
08 899-60-4065 Charles Williams
09 951-05-4521 Jack Velazquez
10 788-35-4977 Donald Cooke
```

It also uses **Privatekey.txt** which holds a passphrase (see *Example 36* on page 113). The following script, **enc_record.fcl**, demonstrates how you can encrypt entire records in the above input file:

```
/INFILE=seqdata_reduced.in
    /FIELD=(WHOLEREC, POSITION=1) # defines the entire record
     # size attribute not required because records are variable-length
/REPORT
/OUTFILE=encrypt_record.out
     # Use passphrase embedded in a limited-access directory file:
    /FIELD=(ENC_WHOLEREC=enc_aes256(wholerec, "file:./Privatekey.txt"), \
            POSITION=1)
```

This produces the file, **encrypt_record.out**:

```
ErAPnA5OOPO76lxs0wFP5gb0XHSyP4Fr93ihBQJmTbY=
9ls86WtCNF05vp7g6LMGcPlHl7Dmw32XQq++n4dB7kc=
0COzkCQweCCNRIG9MIm+0wED9j8+UBUI/8NerhowEv0=
+W2O+3XeKfJ8b5xGnkG0xQM36C+9430cMZAxXnQokCA=
budaT/+B8xCqGXKbNHS/tp6bziHD6KVjFlhq448BUOo=
newFbATPWEvaduk/Pb2FRGytx57HVGcPgQiXScBNLuY=
x2ex/cKzgshYBedfN90ytlv6U89iyouhHgJrMBScUVc=
0wCrAKYgltsR59k1FRJPNSKY08CNXgkv+SiDZAew28o=
ozj4z0DEL/IdiJi6OEL0XuOZ9x2yd6/VovJojUUuebo=
ziSD/xB5qbRAZt+roD+gcLUvRTxgSjvWplSpTNa3qRg=
```

Note that:

- The entire record, indicated by the field wholerec, is encrypted (see ASCII/ALPHANUM Encryption and Decryption *on page 38*).
- The wholerec field was encrypted using a passphrase found in a limited-access directory. See Advanced Encryption Standard (AES) *on page 42* for available options.
- On output, the wholerec field is larger than the original record, and the output records are of a uniform length, due to the encryption process (see Standard Encryption Field Size Considerations *on page 43*). Note that the SIZE attribute is not needed when defining the wholerec field because the records are variable-length.

Example 39 Decrypting an Entire Record

The following script, **dec_record.fcl** decrypts the records that were encrypted using the script **enc_record.fcl** (see *Example 38* on page 115):

```
/INFILE=encrypt_record.out
    # Output file from encrypt operation
    /FIELD=(ENC_WHOLEREC, POSITION=1)
/REPORT
/OUTFILE=restore_record.out
    # Use passphrase embedded in a limited-access directory file:
    /FIELD=(WHOLEREC=dec_aes256(ENC_WHOLEREC, \
      "file:./Privatekey.txt"), POSITION=1)
```

This produces **restore_record.out**:

```
01 775-17-0363 Stuart Clay
02 810-90-1269 Taylor Guerrero
03 906-43-3545 Charles Caldwell
04 787-73-7773 Robyn Puckett
05 950-52-1240 Santiago Lindsey
06 891-56-9799 Charles Lindsey
07 789-12-7387 Santiago Puckett
08 899-60-4065 Charles Williams
09 951-05-4521 Jack Velazquez
10 788-35-4977 Donald Cooke
```

Note that:

- The output records contain the original, restored values (see ASCII/ ALPHANUM Encryption and Decryption *on page 38*).
- A reference to the file **Privatekey.txt** was used to decrypt the record because it holds the same passphrase that was used for encryption.

Example 40 Conditional Field Encryption

The following script, **enc_cond.fcl**, demonstrates how you can apply encryption to one or more fields conditionally. Other fields are left unchanged:

```
/INFILE=seqdata.in
    /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/REPORT
/OUTFILE=encrypt_cond.out
    /CONDITION=(HI_SALARY, TEST=(SALARY2 GE 50000))
    /FIELD=(IDNUM, POSITION=1, SEPARATOR=",")
    /FIELD=(SSNO, POSITION=2, SEPARATOR=",")
    /FIELD=(COND_SALARY2, POSITION=3, SEPARATOR=",", \
           IF HI_SALARY THEN enc_aes256(SALARY2, "ABC") ELSE SALARY2)
    /FIELD=(TRIM_NAME=trim(NAME), POSITION=4, SEPARATOR=",")
```

This produces **encrypt_cond.out**:

```
775-17-0363 Stuart Clay          MQ05u7EB+TTmyYyPLbW0GQ== CT
810-90-1269 Taylor Guerrero      15019.10 MD
906-43-3545 Charles Caldwell     41116.71 NY
787-73-7773 Robyn Puckett        44558.62 NY
950-52-1240 Santiago Lindsey     okAlOvYkYi87UxDlPt2++g== TX
891-56-9799 Charles Lindsey      qL77eS89MHKCA35L6NEUWw== TX
789-12-7387 Santiago Puckett     J3KUMWEZZAdhldo340Z1Yw== NY
899-60-4065 Charles Williams     18645.95 TX
951-05-4521 Jack Velazquez       29205.44 NY
788-35-4977 Donald Cooke         44121.44 MA
```

Note that:

- The salary field is encrypted with 256-bit encryption (see ASCII/ALPHANUM Encryption and Decryption *on page 38*), but only when the value of that field exceeds 50,000. Otherwise, the original field value is displayed (see Conditional Field Security and Masking *on page 19*).
- A literal passphrase was used for encryption. See Advanced Encryption Standard (AES) *on page 42* for available options.
- A `/DATA` statement was used to insert a space before the final field (see /Re-formatting and Masking Options *on page 74*). Note that a `POSITION` attribute was not required for the final field.

To decrypt the encrypted field in a subsequent job script, refer to the example Decrypting Individual Fields *on page 114*.

Example 41 Multiple Field Protections in One Pass

This example, **multi_protect.fcl**, demonstrates the use of multiple field protections in the same I/O pass using a single **FieldShield** job script. It includes pseudonymization, encryption, de-identification, and partial masking.

```
/INFILE=seqdata.in
   /SPECIFICATION=seqdata.ddf # read in metadata from separate .DDF
/REPORT
/OUTFILE=multi_protect.out
      # Invoke look-up table to pseudonymize the name field:
   /FIELD=(FAKE_NAME, SET=pseudo.set[NAME], POSITION=1, SEPARATOR="\t")
      # Use passphrase embedded in a limited-access directory file:
   /FIELD=(ENC_SALARY2=enc_aes256(SALARY2, "file:./Privatekey.txt"), \
          POSITION=2, SEPARATOR="\t")
      # De-Identify the state field. passphrase is 12345678:
   /FIELD=(ID_STATE=de_identify(STATE, "12345678"), POSITION=3, \
           SEPARATOR="\t")
      # Redact the remaining characters of the address, specify size to \
        hide the length:
   /FIELD=(STAR_ADDRESS=replace_chars(ADDRESS, "*", 9), POSITION=4, \
           SEPARATOR="\t", SIZE=20)
```

This produces the output file, **multi_protect.out**:

```
Ramsey Flynn       rOnjXkzwUKWZKfHWGKQd5A==  ?J 101 B St  **********
Clifton Jimenez    S1xieNTw03C4lJBJMqFLvw==  Y@ 1031 Park **********
Teddy Black        9RWOWK8tRQx6bx5kZ+ZXTQ==  z" 14 Main St**********
Salvador Jacobson  0mgUuHHgRyjmRpxZh7pizg==  z" 822 Hwy 76**********
Spencer Craig      jiE9ts3O6hu7mBO6MmIuPQ==  JT Star Rt Bo**********
Landen Sullivan    XIO4d/OvoiwjSjXQF6nPYw==  JT 12746 Wolf**********
Alvaro Mcleod      9qpxI+7YvAwbvGryQmSkMQ==  z" 321 Baltic**********
Jeffery Gomez      cv0ydSoDhYvCSSAprXjSHA==  JT 1103 Fresh**********
Francisco Duffy    ZtnIxDE1mH07Htdl+CeG3Q==  z" 6780 Sand **********
Julio Koch         FIF83PAZLcUcpplIz9ln/A==  Y= 35 La Palm**********
```

Note that:

- The name field in the first column used the file **pseudo.set** to replace original names with their pseudonyms. Stuart Clay has been replaced with Ramsey Flynn, and so on (see Universally Unique Identifier *on page 61*).
- The social security number field in the second column is encrypted with 256-bit encryption, using a passphrase found in a limited-access directory (see ASCII/ALPHANUM Encryption and Decryption *on page 38*). This provides the highest level of field protection.
- The two-character state field in the third column has been de-identified (see Randomization *on page 72*). The key used to de-identify the state field was `"12345678"`. This key must be used in any

subsequent re-identification job (as in *Example 30* on page 106). This provides a relatively low level of protection, and can be restored in a subsequent re-identification job (as in *Example 30* on page 106).

- The address field in the final column has been truncated so that only the first 10 characters appear. This was achieved by using `SIZE=10`, whereas this field was defined with `SIZE=20` on input.
- The final 10 characters of the address field were redacted using a repeated constant string (see /Re-formatting and Masking Options *on page 74*).

Example 42 GPG Encryption

This example demonstrates encryption of data using **enc_gpg()**. Consider the following input file, **personal_info**, which contains credit card numbers, driver's license numbers, and names:

```
9654-4338-8732-8128 W389-324-33-473-Q Jessica Steffani
2312-7218-4829-0111 H583-832-87-178-P Cody Blagg
8940-8391-9147-8291 E372-273-92-893-G Jacob Blagg
6438-8932-2284-6262 L556-731-91-842-J Just Rushlo
8291-7381-8291-7489 G803-389-53-934-J Maria Sheldon
7828-8391-7737-0822 K991-892-02-578-O Keenan Ross
7834-5445-7823-7843 F894-895-10-215-N Francesca Leonie
8383-9745-1230-4820 M352-811-49-765-N Nadia Elyse
3129-3648-3589-0848 S891-915-48-653-E Gordon Cade
0583-7290-7492-8375 Z538-482-61-543-M Hanna Fay
```

The following script, **GPG_encrypt.fcl**, encrypts the name column using the GPG public key. The output file, **personal_info_gpgName**, is shown in **Example 46** as the input file.

```
# GPG cryptology requires other third-party software. See:
# http://www.cryptlib.com/
# http://www.gnupg.org/
#
/INFILE=personal_info
   /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
   /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
   /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE=personal_info_gpgName
   /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
   /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
   /FIELD=(ENC_NAME=enc_gpg(NAME, "public1", "/admin/fsuser/
pubring.gpg"),\
          POSITION=3 , SEPARATOR="\t")
```

The field **name** was encrypted using GPG with the key **public1**.

Example 43 GPG decryption

This example demonstrates decryption of data using **enc_gpg()**. Consider the following input file, **personal_info_gpgName**:

```
9654-4338-8732-8128  W389-324-33-473-Q  wYwDFu5KtyLrpicBBAAgnEpDK...
2312-7218-4829-0111  H583-832-87-178-P  wYwDFu5KtyLrpicBBADBdXznf...
8940-8391-9147-8291  E372-273-92-893-G  wYwDFu5KtyLrpicBBAA7qBKyH...
6438-8932-2284-6262  L556-731-91-842-J  wYwDFu5KtyLrpicBBABybdKe1...
8291-7381-8291-7489  G803-389-53-934-J  wYwDFu5KtyLrpicBBADCZpKOH...
7828-8391-7737-0822  K991-892-02-578-O  wYwDFu5KtyLrpicBBAB4eMT33...
7834-5445-7823-7843  F894-895-10-215-N  wYwDFu5KtyLrpicBBAB3hWvTe...
8383-9745-1230-4820  M352-811-49-765-N  wYwDFu5KtyLrpicBBACXwL3wo...
3129-3648-3589-0848  S891-915-48-653-E  wYwDFu5KtyLrpicBBAC2M7lfQ...
0583-7290-7492-8375  Z538-482-61-543-M  wYwDFu5KtyLrpicBBACSCcZYM...
```

The following script, **GPG_decrypt.fcl**, decrypts the name column using the GPG secret key.

```
# GPG cryptology requires other third-party software. See:
# http://www.cryptlib.com/
# http://www.gnupg.org/
#
/INFILE=personal_info_gpgName
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME, POSITION=3, SEPARATOR="\t")
/REPORT
/OUTFILE=personal_info_restored
    /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR="\t")
    /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR="\t")
    /FIELD=(NAME=dec_gpg(NAME, "sec554", "/admin/fsuser/secring.gpg"),\
           POSITION=3, SEPARATOR="\t")
```

This produces **personal_info_restored**, which restores the original names:

```
9654-4338-8732-8128    W389-324-33-473-Q    Jessica Steffani
2312-7218-4829-0111    H583-832-87-178-P    Cody Blagg
8940-8391-9147-8291    E372-273-92-893-G    Jacob Blagg
6438-8932-2284-6262    L556-731-91-842-J    Just Rushlo
8291-7381-8291-7489    G803-389-53-934-J    Maria Sheldon
7828-8391-7737-0822    K991-892-02-578-O    Keenan Ross
7834-5445-7823-7843    F894-895-10-215-N    Francesca Leonie
8383-9745-1230-4820    M352-811-49-765-N    Nadia Elyse
3129-3648-3589-0848    S891-915-48-653-E    Gordon Cade
0583-7290-7492-8375    Z538-482-61-543-M    Hanna Fay
```

The field **name** was decrypted using GPG with the passphrase **sec554**.

17 Auditing

FieldShield can produce a self-appending log file, in XML format, that contains comprehensive job information for the purposes of auditing. To enable auditing for the execution of your **FieldShield** job script, add this command to the top of the job script:

```
/AUDIT=[path]filename.xml
```

where `[path]` is an optional path (the default path is the working directory from which the job is run). `filename.xml` is the name of the XML log file to contain the audit information. The following are valid `/AUDIT` entries, for example:

```
/AUDIT=FS_log.xml
/AUDIT=usr/admin/logs/FS_log.xml (UNIX/Linux)
/AUDIT=C:\IRI\admin\FS_log.xml (Windows)
```

By default, auditing is not enabled. An audit record is produced for each execution of a job containing the `/AUDIT` statement, and these records append to the XML file specified.

Example 44 on page 123 demonstrates how a **FieldShield** audit record is generated based on the job script contents and user environment at the time of execution. It also shows what the audit file looks like when opened in both a text editor and an XML browser utility.

You can also use **FieldShield** to process, and obtain specific information from, an audit log (see Querying the Audit File *on page 126*).

Supplied DDF File

To facilitate the use of **FieldShield** for generating reports based on audit information, the file **audit_log.ddf** is provided in **$FIELDSHIELD_HOME/etc** (UNIX) or ***install_dir*\etc** (Windows). It includes `/FIELD` references to all the elements contained in the **FieldShield**-generated audit log. It is provided to keep the metadata separate from the application (see Querying the Audit File *on page 126* for an example).

The data elements that comprise the audit log are as follows:

Product	Software product used for job execution, for example `FieldShield`.
Version	Product version number, for example `1.5`.
VersionTag	Product version tag, for example `S070425A-1508`.
Serial	Product serial number, for example `07027.9016`.

OperatingSystem	For example, `Windows XP`.
User	The end user running the job, for example `John_Thomas`.
ProcessId	The operating system process ID, for example `3992`.
Terminal	The ID of the **FieldShield** user's connection to the host.
Program	Path name and file name of the executable used to perform the job, such as `c:\Work_area\fieldshield`.
Command	References the command line entry that launched the job, which specifies the job script name, for example `/spec=csaudit.fcl`.
StartTime	Start time and date of **FieldShield** job execution, in ISO_TIMESTAMP format (see *Table 35* on page 234), for example `2008-01-19 21:47:15`.
EndTime	End time and date of **FieldShield** job execution in ISO_TIMESTAMP format, for example, `2008-01-19 21:49:06`.
RunTime	The runtime duration of the **FieldShield** job (the difference between StartTime and EndTime) in HH:MM:SS format, for example `00:01:51`.
ReturnCode	For example, `0`, if the job completed without error.
ErrorMessage	The error message associated with job execution. `normal return` indicates no error.
RecordsProcessed	The number of input records processed (after any pre-processing record filter logic was applied).
Script	The complete text of the job script, where each line of entry is separated by a space.
Environment	All environment variables at the time of execution, which will show the literal equivalents of any environment variables referenced in the job script.

Example 44 Creating an /AUDIT File

Consider that the following entry is set in a **FieldShield** job script:

```
AUDIT=C:\FieldShield\Tests\mytests\audit\csaudit.xml
```

Consider that the following **FieldShield** job script, **audit.fcl**, is executed:

```
/INFILE=$CSINPUT
    /SPECIFICATION=fields.ddf
/REPORT
/OUTFILE=$CSOUTPUT
    /SPECIFICATION=fields.ddf
```

where **fields.ddf** contains the first two `/FIELD` statements and a nested `/SPEC` entry that refers to the remaining two fields of the input record:

```
/FIELD=(name,POSITION=1,SIZE=27,ASCII)
/FIELD=(year,POSITION=28,SIZE=12,ASCII)
/SPECIFICATION=fields2.ddf
```

and where **fields2.ddf** contains:

```
/FIELD=(party,POSITION=40,SIZE=5,ASCII)
/FIELD=(state,POSITION=45,SIZE=2,ASCII)
```

When the above job script is executed, an entry in the audit file **csaudit.xml**, is created/added. When **csaudit.xml** is opened in a text editor, the **.ddf** file names and their components are tabbed within the `<Script>` element to allow for easier readability of nested script specifications, as follows:

```
...
<Script>
/AUDIT=C:\FieldShield\Tests\mytests\audit\csaudit.xml
/INFILE=C:\FieldShield\Tests\mytests\audit\seqdata.in
    /SPECIFICATION=fields.ddf
    /FIELD=(name,POSITION=1,SIZE=27,ASCII)
    /FIELD=(year,POSITION=28,SIZE=12,ASCII)
        /SPECIFICATION=fields2.ddf
            /FIELD=(party,POSITION=40,SIZE=5,ASCII)
            /FIELD=(state,POSITION=45,SIZE=2,ASCII)
/OUTFILE=C:\FieldShield\Tests\mytests\audit\seqdata.out
    /SPECIFICATION=fields.ddf
    /FIELD=(name,POSITION=1,SIZE=27,ASCII)
    /FIELD=(year,POSITION=28,SIZE=12,ASCII)
        /SPECIFICATION=fields2.ddf
            /FIELD=(party,POSITION=40,SIZE=5,ASCII)
            /FIELD=(state,POSITION=45,SIZE=2,ASCII)
</Script>
...
```

However, when opened in an XML-supported browser, the `<script>` entry displays as a series of space-delimited entries and there are no tabs. An XML browser will therefore display **csaudit.xml** as follows (note that the `<Environment>` entry near the bottom has been truncated for readability purposes):

```
 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <AuditTrail>
- <AuditRecord>
  <Product>FieldShield</Product>
  <Version>2.1</Version>
  <VersionTag>D91090618-1454</VersionTag>
  <Serial>12345.6789</Serial>
  <OperatingSystem>Windows XP</OperatingSystem>
  <User>fieldshield_user</User>
  <ProcessId>4784</ProcessId>
  <Program>C:\FieldShield\Tests\mytests\audit\fieldshield</Program>
  <Command>/sp=audit.fcl</Command>
  <StartTime>2008-01-19 21.47.15</StartTime>
  <EndTime>2008-01-19 21.48.06</EndTime>
  <RunTime>00:01:51</RunTime>
  <ReturnCode>0</ReturnCode>
  <ErrorMessage>normal return</ErrorMessage>
  <RecordsProcessed>42</RecordsProcessed>
  <Script>/AUDIT=C:\FieldShield\Tests\mytests\audit\csaudit.xml
/STATITICS=C:\FieldShield\Tests\mytests\audit\csstat.log
/INFILE=C:\FieldShield\Tests\mytests\audit\seqdata.in
/SPEC=fields.ddf /FIELD=(name,POSITION=1,SIZE=27,ASCII)
/FIELD=(year,POSITION=28,SIZE=12,ASCII)
/SPEC=fields2.ddf
/FIELD=(party,POSITION=40,SIZE=5,ASCII)
/FIELD=(state,POSITION=45,SIZE=2,ASCII)
/OUTFILE=C:\FieldShield\Tests\mytests\audit\seqdata.out
/SPEC=fields.ddf
/FIELD=(name,PPOSITION=1,SIZE=27,ASCII)
/FIELD=(year,POSITION=28,SIZE=12,ASCII)
/SPEC=fields2.ddf /FIELD=(party,POSITION=40,SIZE=5,ASCII)
/FIELD=(state,POSITION=45,SIZE=2,ASCII)</Script>
  <Environment>ALLUSERSPROFILE=C:\Documents and Settings\
All Users APPDATA=C:\Documents and Settings\fieldshield_user\Application Data
CLASSPATH=.;C:\Program Files\Java\j2re1.4.2_03\lib\ext\QTJava.zip
CLIENTNAME=Console CommonProgramFiles=C:\Program Files\Common Files
...
</Environment>
</AuditRecord>
</AuditTrail>
```

NOTE

This audit file contains only one record. Subsequent jobs that are written to **csaudit.xml** are appended to the bottom, and always begin with a new `<AuditRecord>` tag.

Example 45 Querying the Audit File

You can use `/PROCESS=XML` in the input section of a **FieldShield** job script to process the records contained in a **FieldShield** audit log (see XML *on page 180*).

Consider the following XML audit log, **catalog.xml**, which was generated by **FieldShield** (note that the `<Environment>` entry has been truncated for readability purposes):

```
  <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <AuditTrail>
- <AuditRecord>
  <Product>FieldShield</Product>
  <Version>1.5</Version>
  <VersionTag>D91090618-1454</VersionTag>
  <Serial>12345.6789</Serial>
  <OperatingSystem>Windows 2000</OperatingSystem>
  <User>Bill_Evans</User>
  <ProcessId>1792</ProcessId>
  <Terminal>console</Terminal>
  <Program>C:\IRI\FieldShield11\bin\fieldshield</Program>
  <Command>/spec=catalog_plant.fcl</Command>
  <StartTime>2008-01-12 09.59.27</StartTime>
  <EndTime>2008-01-12 10.01.17</EndTime>
  <RunTime>00:01:50</RunTime>
  <ReturnCode>0</ReturnCode>
  <ErrorMessage></ErrorMessage>
  <RecordsProcessed>0</RecordsProcessed>
  <Script>/INFILE=plant_catalog.txt /PROCESS=record
/FIELD=(common, POSITION=1, SEPARATOR='^')
/FIELD=(botanical, POSITION=2, SEPARATOR='^')
/FIELD=(zone, POSITION=3, SEPARATOR='^')
/FIELD=(light, POSITION=4, SEPARATOR='^')
/FIELD=(price, POSITION=5, SEPARATOR='^')
/FIELD=(availability, POSITION=6, SEPARATOR='^')
/OUTFILE=catalog_plant.xml</Script>
  <Environment>ALLUSERSPROFILE=C:\Documents and Settings\
All Users.WINNT APPDATA=C:\Documents and Settings\Bill_Evans\Application
Data CommonProgramFiles=C:\Program Files\Common Files COMPUTERNAME=ATHENA
...
  </Environment>
  </AuditRecord>
  </AuditTrail>
```

NOTE

For the purposes of this example, the complete **catalog.xml** is comprised of several **FieldShield** jobs (audit records), the first of which is shown above.

If you want to perform detailed analysis on your audit log, such as sorting and 'group by' aggregation, it is recommended that you upgrade to **CoSort**, which uses the same job script syntax and metadata constructs as **FieldShield** (see the Upgrade Options *chapter on page 215*).

The following script, **catalog.fcl**, reads the data in the XML audit file, **catalog.xml** and outputs records in CVS format that can be read using a spreadsheet utility such as Microsoft® Excel® (see CSV *on page 172*):

```
/INFILE=audittrail.xml
    /PROCESS=XML
    /SPECIFICATION=audit_log.ddf
/OUTFILE=csaudit.csv
    /PROCESS=CSV
    /INCLUDE WHERE COMMAND CT "catalog" # job script contains this name
    /FIELD=(USER, POSITION=1, SEPARATOR=",", FRAME='"')
    /FIELD=(COMMAND, POSITION=2, SEPARATOR=",", FRAME='"')
    /FIELD=(STARTTIME, POSITION=3, SEPARATOR=",", FRAME='"')
```

This produces **csaudit.csv**, which might look like this for example (assuming the input source, **catalog.xml**, consists of several records):

```
User,Command,StartTime,StartDate
"Bill_Evans","/specification=catalog_plant.fcl","2008-01-12 09.59.27"
"Dave_Johnston","/specification=catalog_survey.fcl","2008-01-10 08.16.13"
"John_Thompson","/specification=catalog_plant.fcl","2008-01-14 10.35.23"
"Marcy_Wallace","/specification=catalog_survey.fcl","2008-01-08 07.23.54"
"Roger_Smith",",","/specification=catalog_report.fcl","2008-01-10 22.59.22"
```

The audit file was processed, and only the desired output records (those with `catalog` in the job script name) were produced in CSV format.

NOTE

Using `/PROCESS=XML` on input, and referring to the input `/FIELD` specifications provided in **audit_log.ddf**, you can query and analyze the audit log file using any of the **FieldShield** options described throughout this chapter such as record filter logic, field-level IF THEN ELSE logic, and reformatting.

Dynamic Data Masking

1 Overview

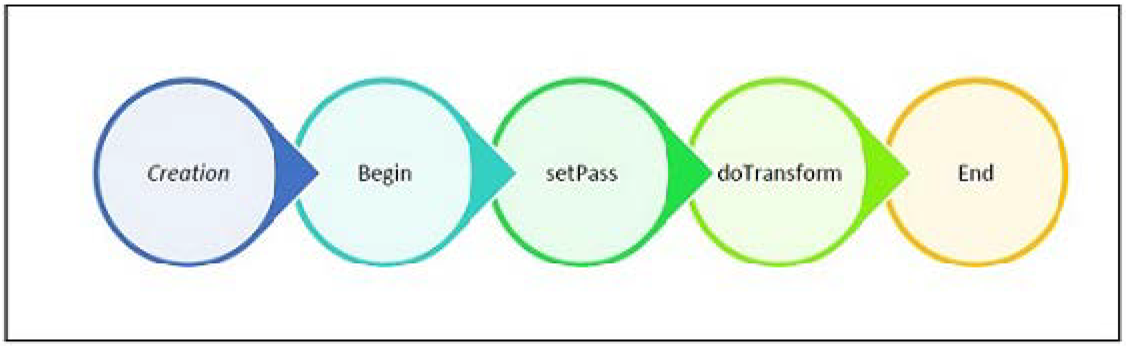
The software development kit (SDK) for dynamic data masking (DDM) within applications uses APIs written in Java and .NET. The FieldShield library is a group of classes that provides several ways of protecting or altering data by using encryption/ decryption processes, hashing, masking, and base 64 to hex conversion.

The FieldShield SDK's encryption and decryption algorithms include:

- AES-256 (Advanced Encryption Standard); the state-of-the-art and most powerful of the encryption algorithms.
- FP-ASCII (Format Preserved for American Standard Code for Information Interchange); the original format data is preserved but still protected with AES-256 bit encryption.
- FP- ALPHANUM; another version of the Format Preserving algorithm above, in which it can be set to number mode only or alphanumeric for data preservation.

1.1 FieldShield Library Process

Each function in the FieldShield library flows in a defined manner. The figure below shows the sequence of methods for each encryption and decryption usage. Once the password is set for a given object, it does not need to be set each time the encryption object is used.



The steps are defined as follows:

1. Creation
   In this step, create an instance of the desired class. Most languages use the keyword **new**.

2. Begin

After creating an instance from the class, use this instance to call the **begin()** method. This function will initiate the encryption or decryption process.

3. SetPass

Next, the pass phase must be set by calling the **setPass** method. This method requires a String to be passed in as a parameter.

4. doTransform

Once the passphrase has been set, the core method, **doTransform(Data)**, is called to encrypt or decrypt. A String is passed as a parameter, which is then encrypted or decrypted using the algorithm chosen. For the alphanumeric class, a numeric **doTrasform** method exists to be used on numerals. The method will return the data that has been encrypted or decrypted with the appropriate type.

5. destruct

Finally, call the method **destruct()** to release the object from the occupied space in memory. Since it uses the DLL file, it is necessary to destruct the object.

1.2 FieldShield Library Main Classes

The three types of available encryption/decryption algorithms include AES256, FPASCII, and FP-ALPHANUM. Each type has two classes, one for encryption and the other for decryption. The classes are:

AES256

- enc_aes256: AES256 – Encryption
- dec_aes256: AES256 – Decryption

FP – ASCII

- enc_fp_ascii: FP- ASCII – Encryption
- dec_fp_ascii: FP- ASCII – Decryption

FP – ALPHANUM

- enc_fp_alphanum: FP- ALPHANUM – Encryption
- dec_fp_alphanum: FP- ALPHANUM – Decryption

The other classes available are for hashing, changing to base 64 or hex, and replacing characters in a String. The classes are:

- hex: Hex conversion
- hash_sha2: Hashing based on the SHA256
- base64: Base 64 conversion
- replace_char: Replacing characters (masking)

2 Java

In the following instructions, the Eclipse IDE was used when implementing the Java objects for the API. Other IDEs will vary slightly.

1. Download the FieldShield Library. This includes two library files:
   - libcscrypt.dll
   - encJava.jar
2. Set up the libraries.
   This library has the core functions coded in .NET. This file is native library which is to be used in java. There are two ways to use this library:

2.1 Command line

To set the java library path through the command line, use the following command. This must be set so that the JNI can locate the .NET SDK to perform the operations. The path must be the location of the folder that contains the DLL file of .NET SDK of encryption project.

Java run configuration's arguments : VM arguments:

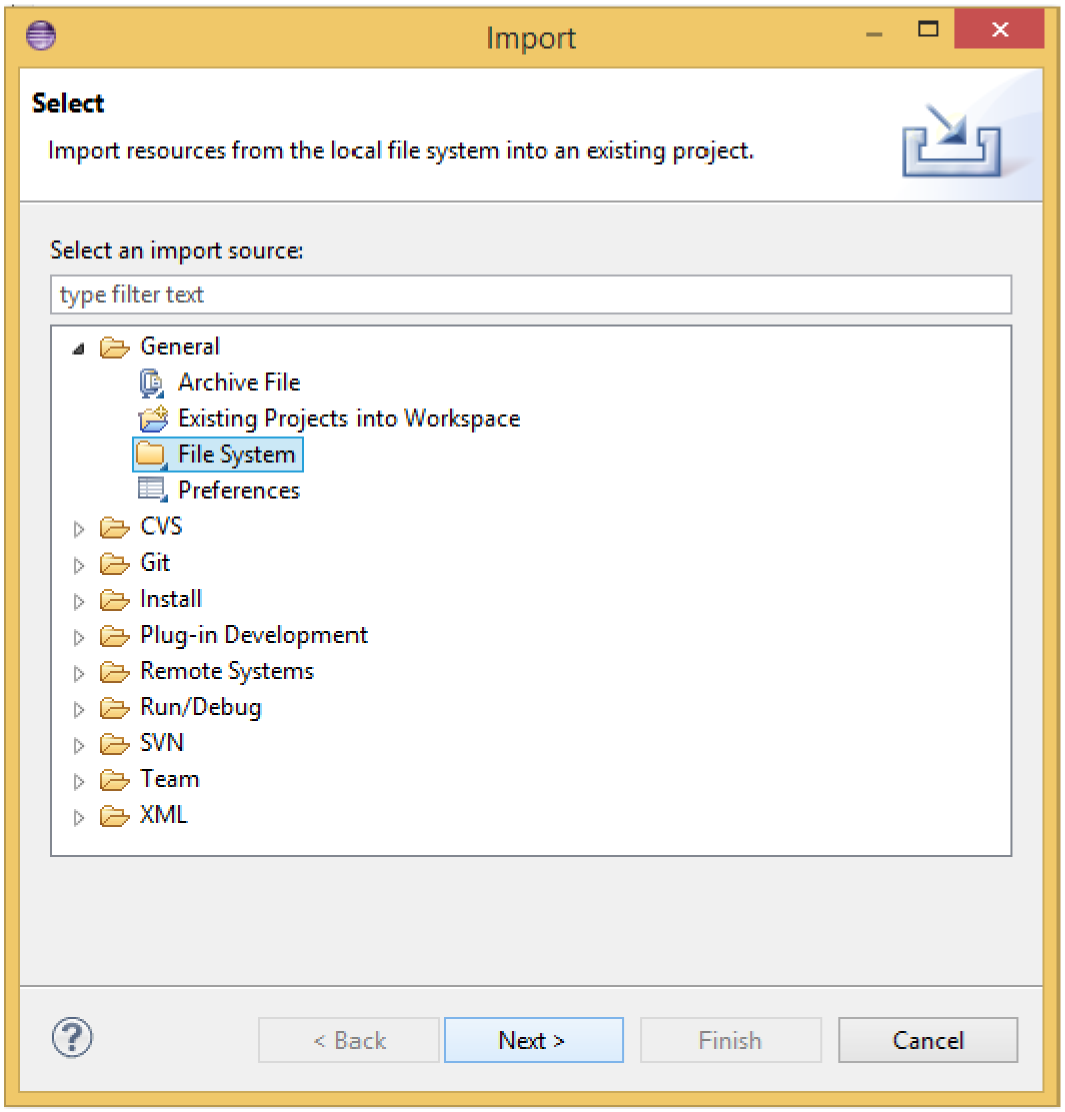
```
java -Djava.library.path=C:/projects/libcscrypt/libcscrypt/Debug TestApp
```

2.2 Eclipse

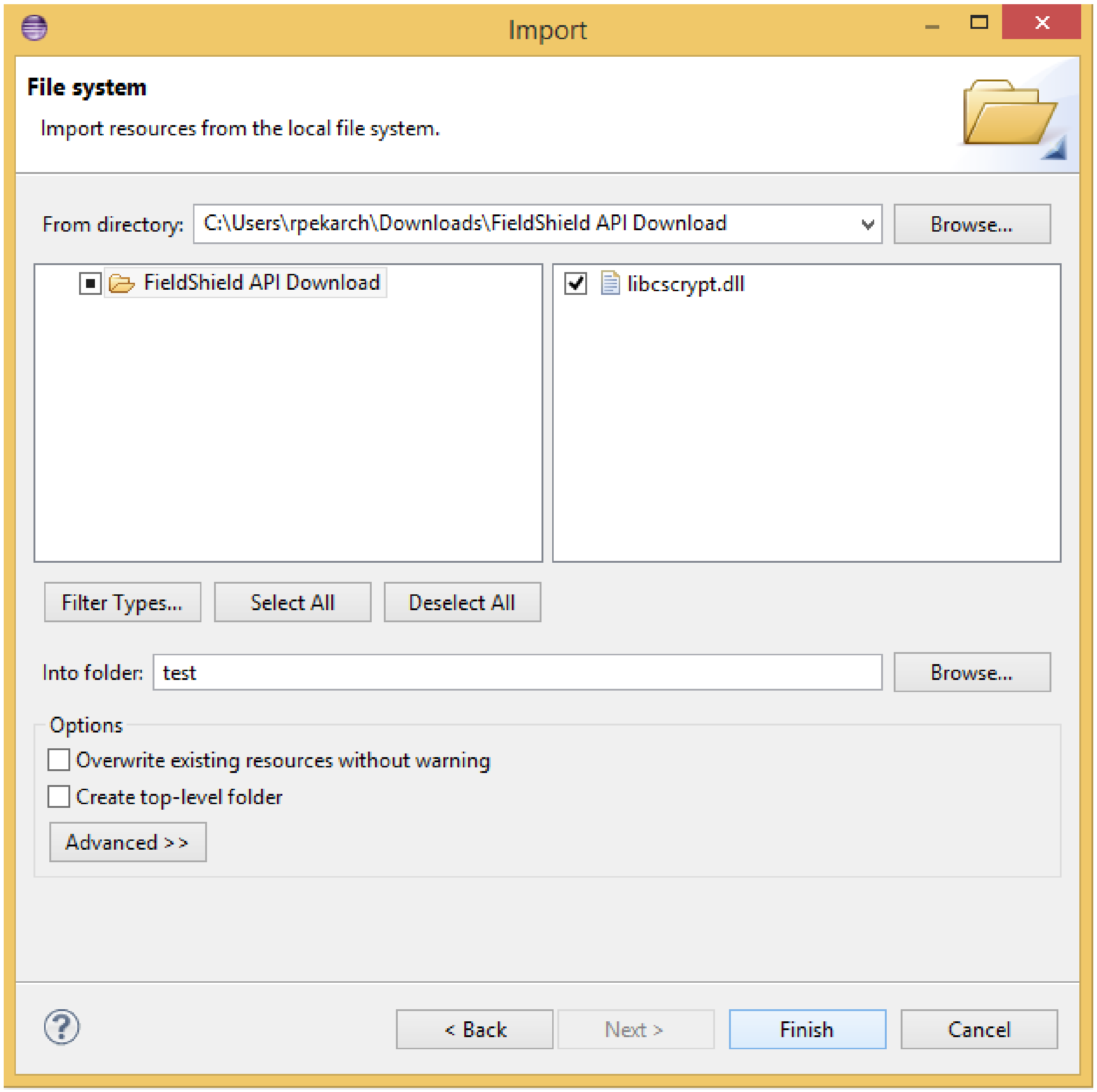
Import the files into a Project for the Eclipse IDE

Both the jar file and DLL file must be imported into the project in which the classes are to be used.

1. Copy the DLL file into the Project folder of your eclipse java project, or right-click the project and select **Import**. On the Import page, select **General**, and the select **File System**. Click **Next**.



2. On the File System page, browse for the directory with the libcscrypt.dll, then select that DLL. Click **Finish**. The libcscrypt.dll is now included in the Project folder.



From directory: C:\Users\rpekarch\Downloads\FieldShield API Download

FieldShield API Download

[✓] libcscrypt.dll

Into folder: test

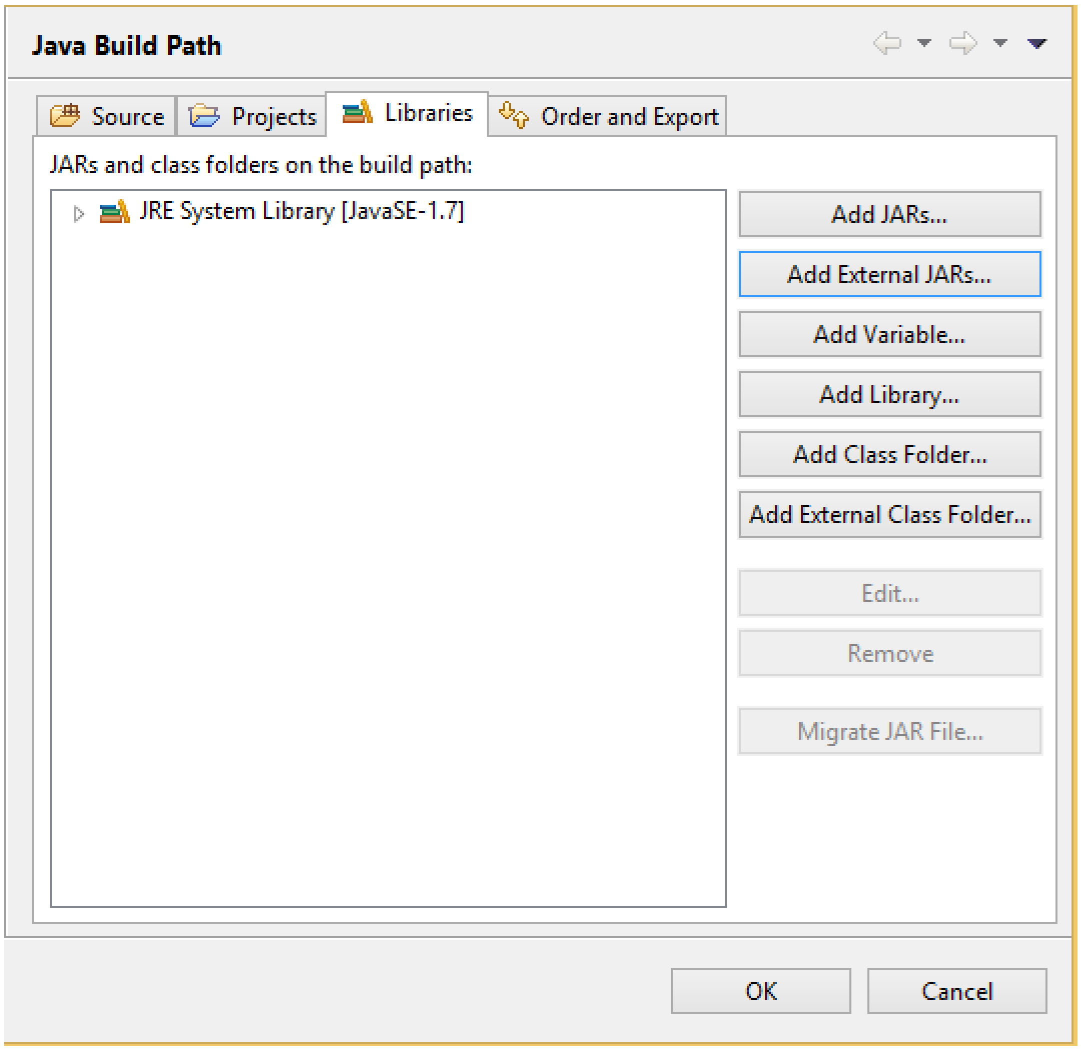
[ ] Overwrite existing resources without warning

[ ] Create top-level folder

Add the jar to the Project

1. Right-click on the project and select **Build Path**. Select either **Add External Archives** or **Configure Build Path**. The Properties page open. Go to the Libraries tab.

2. Click **Add External Jars** and find and select the downloaded jar file. Click **Open**.



3. Select **encJava.jar** and click **OK**. The jar will be included under the libraries folder in the project. In order to use the classes in the project, you must import **UtilityClass.*** Refer to the API documentation to create and use objects of these classes, or see the included examples.

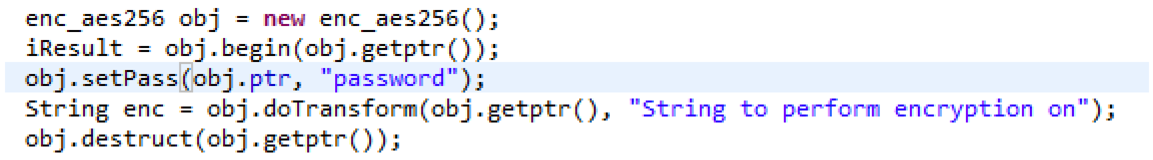
NOTE The JAR file works on the functions defined in the .NET DLL file that has the core code of the encryption/decryption operations. The latest .NET DLL version must be replaced to be able to use the updated/extended functions in java.

2.3 FieldShield Library Usage

In order to use the methods in Java, you must do several steps to create the object to be passed to the DLL. Following is the code sample to use the AES256 Encryption.

Once the object is instantiated, pass the pointer to the begin() method. Then set the password with the setPass method, providing the arguments of the pointer and String that is to be the password. Use the doTransform method by passing the pointer and the String (or double if it’s the alphanumeric version for numbers). This method will return the encrypted String. Finally, you must deallocate the object by passing the pointer to the destruct or end method. The other two encryptions work in the same way.

Java Example



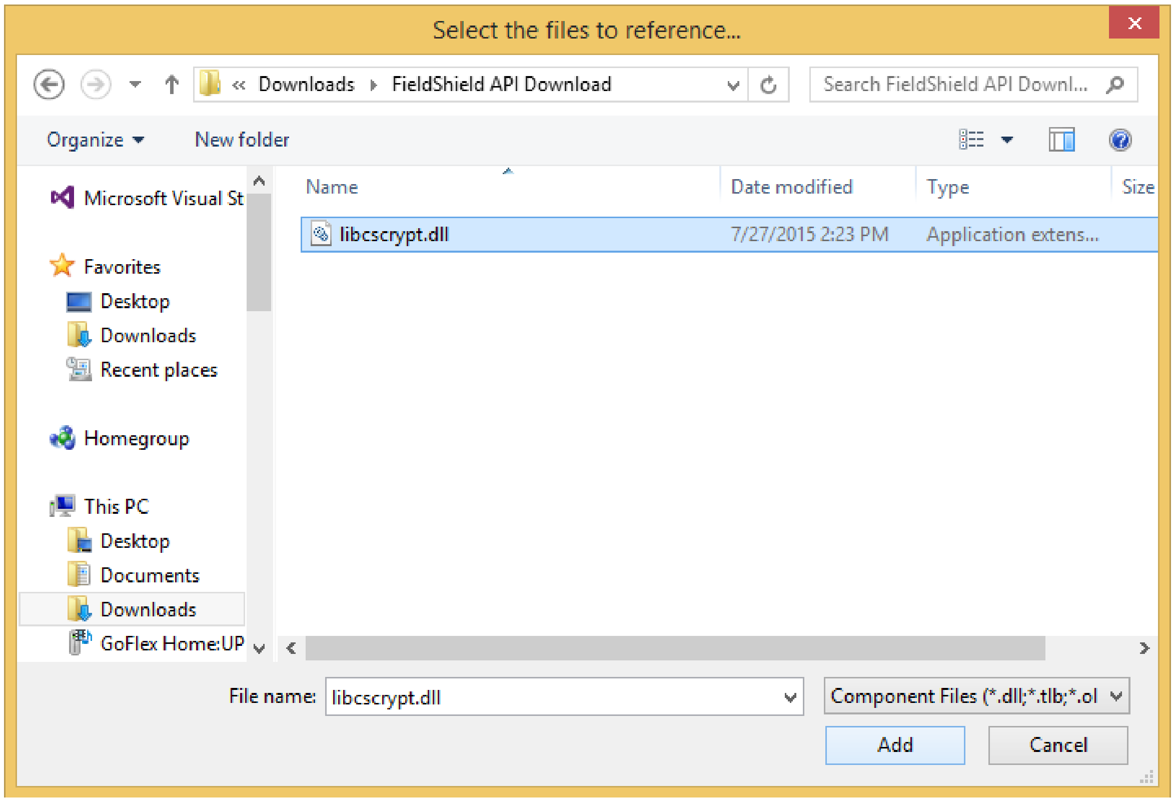
```
enc_aes256 obj = new enc_aes256();
iResult = obj.begin(obj.getptr());
obj.setPass(obj.ptr, "password");
String enc = obj.doTransform(obj.getptr(), "String to perform encryption on");
obj.destruct(obj.getptr());
```

3 Microsoft Visual Studio

Visual Studio 2013 was used when implementing the .NET objects for the API. Any of the languages supported in Visual Studio are capable of using the FieldShield SDK. Support documentation has been done for C++,C#, F# and Visual Basic. If using a language other than the ones documented, refer to the language’s syntax in order to use the library.

3.1 Installing the FieldShield Library

1. Download the FieldShield Library: libcscrypt.dll
2. From MicroSoft Visual Studio, open the project into which the DLL will be imported.
3. From the Solution Explorer window, right-click the project to add the reference, and select **Add** -> **Reference.** The Reference Manager opens.
4. Click **Browse** in the directory with the DLL. Select the DLL, and then click **Add**.

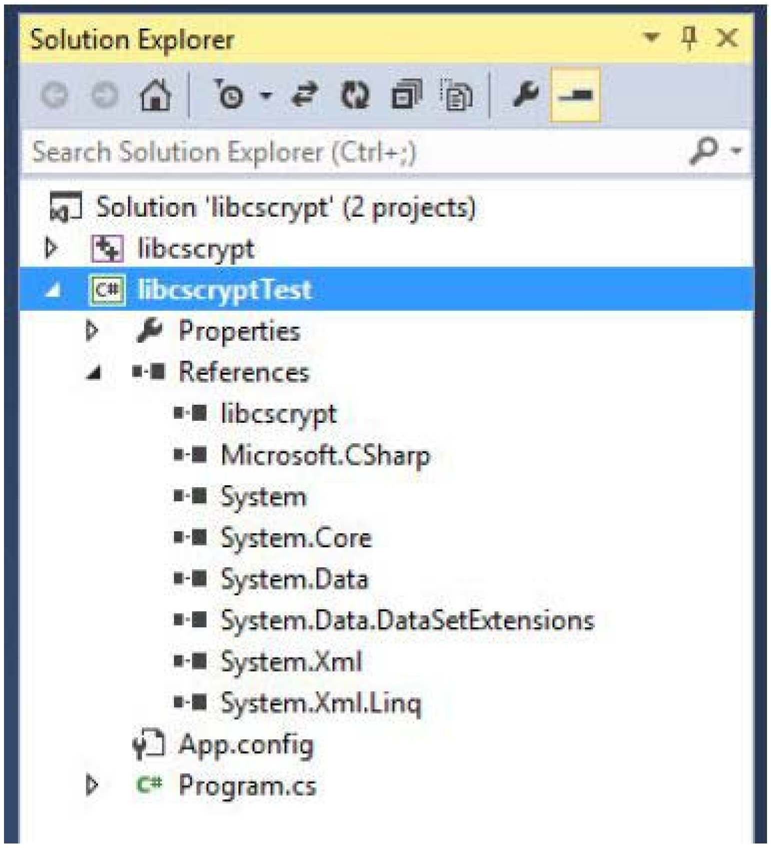


Name	Date modified	Type	Size
libcscrypt.dll	7/27/2015 2:23 PM	Application extens...	

File name: libcscrypt.dll

Component Files (*.dll;*.tlb;*.ol

5. Click **OK** to add the reference. It will display in Solution Explorer under the References section. Check to make sure the DLL is added. Once this is done, use the proper import keyword for the particular .NET language in which the methods will be used.



Solution 'libcscrypt' (2 projects)
libcscrypt
libcscryptTest
Properties
References
libcscrypt
Microsoft.CSharp
System
System.Core
System.Data
System.Data.DataSetExtensions
System.Xml
System.Xml.Linq
App.config
Program.cs

The following is the code sample to use the AES256 Encryption in C#. The other classes work in a similar manner. The HTML help file can be referenced for specific implementation in another language.

.NET Example in C#

```
enc_aes256 encryptAES256 = new enc_aes256();
iResult = encryptAES256.begin();
encryptAES256.setPass("password");
string test1 = encryptAES256.doTransform("String to perform encryption on");
encryptAES256.end();
```

FieldShield Language Basics

1 Files

This section describes the types of files that can be referenced by **FieldShield**:

- Data Files *on page 139*
    - Input Files *on page 139*
    - Output Files *on page 142*
- Data Definition Files *on page 143*
- Specification Files *on page 144*

1.1 Data Files

This section describes the standard **FieldShield** data sources and targets: input files and output files.

An input or output file description consists of a file name and set of attributes such as process type, fields, and filters.

1.1.1 Input Files

You must name the input source with an `/INFILE` statement:

```
/INFILE=[path]filename
```

Statements that describe the input source are then placed under the `/INFILE` statement. Note that **stdin** (standard input) is a valid input file name when records are to be piped into **FieldShield**.

For an example of the `/INFILE` statement, consider records from the input file **miami**:

```
Reasoning For  Prentice-Hall 150     10.25
Murder Plots   Harper-Row    160      5.90
Still There    Dell           80     13.05
Pressure Cook  Harper-Row    228      9.95
```

which can be referenced within your script as follows:

```
/INFILE=miami                                # names the input file
    /FIELD=(title,POSITION=1,SIZE=15)        # book title
    /FIELD=(publisher,POSITION=16,SIZE=13)# book publisher
    /FIELD=(quantity,POSITION=30,SIZE=3)   # number of books sold
    /FIELD=(price,POSITION=38,SIZE=5,NUMERIC) # price of book
```

NOTE If an input file is empty, an empty output file is created.

Spaces within File Names/Paths -- Windows Users Only

W In some cases, you may want **FieldShield** to reference a path name or file name that contains a space. **FieldShield** supports the use of a space (but not two or more in a row) within a path component and/or file name reference. This feature applies to the statements `/INFILE` (see Input Files *on page 139*), `/OUTFILE` (see Output Files *on page 142*), `/SPEC` (see Specification Files *on page 144*), and `/FILE` (see Data Definition Files *on page 143*). An example is:

```
/INFILE=C:\iri\fieldshield11\test file1
```

Multiple Input Files (INFILES)

If you want to process more than one input file whose layouts are identical, you can use the following syntax to define the files:

/INFILES=([*path*]*filename1*,[*path*]*filename2*,...)

In this case, you must define the common record layout beneath the `/INFILES` statement, and the multiple files will be processed together as a single source, for example:

```
/INFILES=(seqdata.in,seqdata2.in)
    /FIELD=(idnum,POSITION=1,SIZE=2.0,NUMERIC
    /FIELD=(ssno,POSITION=4,SIZE=11)
    /FIELD=(name,POSITION=16,SIZE=18)
    /FIELD=(salary,POSITION=34,SIZE=8,NUMERIC)
    /FIELD=(deduction_no,POSITION=43,SIZE=1)
    /FIELD=(state,POSITION=45,SIZE=2)
    /FIELD=(address,POSITION=48,SIZE=20)
```

NOTE If you have multiple input files with disparate formats, and you want to process them together in one job script, you can upgrade to **CoSort**. Contact IRI for more information.

Wildcards used with /INFILES

You may specify wildcard characters such as `*` and `?` for your `/INFILES` sources.
In this case, **FieldShield** expands the file name and uses the expanded name during transformation. When defining your `/INFILES` attributes, you must ensure that they pertain to all files referenced by the wildcard expansion (see above for
`/INFILES` syntax). If there is no matching file name available, **FieldShield** terminates with the error `No such file`. For example, if you have a directory containing the files **chiefs**, **thiefs**, **chiefs_sep**, and you want all of these to be referenced by your `/INFILES` statement, you can use:

```
/INFILES=?hiefs*
```

Using gzip Format as an INFILE source

FieldShield supports input files in compressed gzip format. Deflated files are automatically recognized by the unique gzip header information at the beginning of the input file.

To process an input file in gzip format, name the gzip (deflated) file in the **/INFILE** line. You can mix regular and deflated files in the same **/INFILES** section, as each file is scanned separately.

NOTE `HEADSKIP` and `/TAILSKIP` are not supported when using gzip input sources. Results will be unpredictable if you use either of these statements.

/INPROCEDURE

You can invoke your own custom input procedures and link them to **FieldShield** to accommodate special job criteria (see Custom Input *on page 187*). All custom input routines are invoked using `/INPROCEDURE`, and attributes are provided directly beneath this statement.

1.1.2 Output Files

As input records are processed through **FieldShield**, they can be mapped into one or more output files. A possible output target is also **stdout** (standard out), which, by default, sends results to the monitor.

Output files are declared with one or more `/OUTFILE` statements:

```
/OUTFILE=[path]filename
```

Further statements for describing the file are placed beneath this statement. You may define multiple output files with a new `/OUTFILE` statement for each, for example:

```
/OUTFILE=apples
   attributes
/OUTFILE=peaches
   attributes
/OUTFILE=pears
   attributes
```

See *Example 27* on page 103 for an example of producing multiple output files.

Remapping

You can assign field positions and their sizes in the output section differently than the way they were specified in the input section (see POSITION *on page 148* and SIZE *on page 153*). For an example of remapping, see *Example 26* on page 102.

NOTE

When an input file is empty, an empty output file is created.

Using gzip Format as an OUTFILE target

FieldShield supports the writing of output files in compressed gzip format. To produce an output file in gzip format, give the filename a **.gz** or **.gzip** extension in the `/OUTFILE` entry, and the output file will be deflated and stored in gzip format.

/OUTPROCEDURE

You can invoke your own custom output procedures and link them to **FieldShield** to accommodate special job criteria (see Custom Output *on page 188*). All custom output routines are invoked using `/OUTPROCEDURE`, and attributes are provided directly beneath this statement.

1.2 Data Definition Files

For data sharing purposes, it is recommended that you develop and use a specification file that contains data definitions only. This type of metadata repository is referred to as a *Data Definition File* (DDF). It contains a `/FILE` statement followed by further attributes for the named data file. The file described can be used as an input or an output file, and are an ideal way to create user views and share common data.

NOTE

Several **FieldShield** tools convert third-party metadata into **FieldShield** data definition files. See the FieldShield Metadata Tools *chapter on page 193*.

You can also make use of **MIMB**, a gateway application that converts metadata from one application format into another. Developed by Meta Integration Technology, Inc. (MITI), **MIMB** provides built in metadata management and conversion support for IRI products like **CoSort**, **RowGen** and **FieldShield** (see the Upgrade Options *chapter on page 215*).

When an `/INFILE` or `/OUTFILE` statement occurs in a job specification file, the record definitions may be obtained from the same-named `/FILE` layout within the data definition file. The data definition file reference must appear before any `/INFILE` or `/OUTFILE` that will reference the layouts in that data definition file.

For example, the data definition files **chiefs.ddf** and **parties.ddf** could contain:

```
# chiefs.ddf
    /FIELD=(president,POSITION=1,SIZE=22)
    /FIELD=(votes,POSITION=24,SIZE=3)
    /FIELD=(service,POSITION=28,SIZE=9)
    /FIELD=(party,POSITION=40,SIZE=3)
    /FIELD=(state,POSITION=45,SIZE=2)
```

```
# parties.ddf
    /FIELD=(party,POSITION=5,SIZE=3)
    /FIELD=(president,POSITION=10,SIZE=22)
```

and the job specification file could be:

```
/INFILE=chiefs
    /SPECIFICATION=chiefs.ddf
/REPORT
/OUTFILE=parties
    /SPECIFICATION=parties.ddf
```

1.3 Specification Files

These files are used to organize **FieldShield** statements. These named files are referenced by the `/SPECIFICATION` (or `/SPEC`) statement as shown below. One or more specification files may be used in the execution of the **FieldShield** program, for example:

```
fieldshield /SPECIFICATION=[path]file_1  /SPECIFICATION=[path]file_2 ...
```

A specification file, or *script*, may have other specification files referenced within it, and these may be nested as deeply as necessary. The following diagram demonstrates three levels of nesting.

```
SpecFile_1
  FS statement
  FS statement
  /SPEC=SpecFile_2  ───►  SpecFile_2
  FS statement              FS statement
        ...                 FS statement
        ...                 /SPEC=SpecFile_3  ───►  SpecFile_3
        ...                 FS statement              FS statement
        ...                      ...                  FS statement
        ...                      ...                       ...
  FS statement                                             ...
  /SPEC=SpecFile_4  ───►  SpecFile_4
                            FS statement
                            FS statement
                                 ...
                                 ...
```

For discussion purposes, we will refer to the *job specification file* as the principal group (or first level) of statements.

When a `/SPECIFICATION=`*filename* occurs within a specification file, the contents of the referenced file are read into the job at that point.

For example, consider a file named **ID.ddf** that contains the following:

```
/FIELD=(Name,POSITION=1,SIZE=20)
/FIELD=(Age,POSITION=21,SIZE=3)
```

Consider a job specification file written as follows:

```
/INFILE=IDs.in
    /SPECIFICATION=ID.ddf
/REPORT
/OUTFILE=IDs.out
```

This is the equivalent of writing the following job specification:

```
/INFILE=IDs.in
    /FIELD=(Name,POSITION=1,SIZE=20)
    /FIELD=(Age,POSITION=21,SIZE=3)
/REPORT
/OUTFILE=IDs.out
```

Note that instead of using the name of a specification file, you may use an environment variable that references the file. In that case, the job specification file might be as follows:

```
/INFILE=IDs.in
    /SPECIFICATION=$IDfields
/REPORT
/OUTFILE=IDs.out
```

2 Fields

Data files consist of records; records consist of fields. The location of a field is either:

- fixed position, where the starting byte for a field is always in the same column
- delimited, where the starting position for any field after the first field is to the right of a separator character.

In both cases, numbering begins with *one* (1). **FieldShield** allows you to have both fixed and floating fields in the same input record. The defined fields may also overlap each other in an input record.

2.1 Syntax: Input /FIELD Definitions

The following is the syntax for defining fixed-position input fields:

```
/FIELD=(fieldname,[,TYPE=data type] POSITION=n,SIZE=n[.n]
[,PRECISION=n][,FRAME='char'][,alignment]
[,FILL='char'][,XDEF=element][,EXT_Field=filename])
```

The following is the syntax for defining delimited input fields:

```
/FIELD=(fieldname,[,TYPE=data type] POSITION=n,SEPARA-
TOR='char(s)'[,SIZE=n[.n]][,PRECISION=n][,FRAME='char']
[,alignment][,FILL='char'][,XDEF=element])[,EXT_Field=filename]
```

Field attributes are separated by commas. The field name must appear first, but other attributes can appear in any order. The `n` shown for some of the parameter values represents a whole number.

NOTE The XDEF attribute is applicable only when using XML files (see XML *on page 180*).

2.2 Syntax: Output /FIELD definitions

There are additional supported attributes when defining `/FIELD`s for output targets.

The following is the syntax for defining fixed-position output fields:

```
/FIELD=(fieldname [,TYPE=data
type][,SET=filename],POSITION=n,SIZE=n[.n]
[,PRECISION=n][,FRAME='char'][,alignment][,FILL='char']
[,XDEF=element][,IF-THEN-ELSE])[,EXT_Field]
```

The following is the syntax for defining delimited output fields:

```
/FIELD=(fieldname[,TYPE=data type][,SET=filename],
POSITION=n,SEPARATOR='char(s)'[,SIZE=n[.n]][,PRECISION=n]
[,FRAME='char'][,alignment][,FILL='char'][,XDEF=element]
[,IF-THEN-ELSE])[,EXT_Field]
```

NOTE

The `SET` attribute is described in Universally Unique Identifier *on page 61*.

`IF-THEN-ELSE` logic is described in Conditional Field Security and Masking *on page 19*.

2.3 Field Name

The field name identifies the field and can be referenced in subsequent **FieldShield** statements.

In the output target definition only, it is possible to have a field definition consisting of only an input field name. That is, without other attributes such as `POSITION`. Each defined field will be mapped sequentially (one after the other). For example, if you had defined the fields `lastname` and `firstname` in the input file, you could have the following field definitions for the output file:

```
/FIELD=(lastname)
/FIELD=(firstname)
```

Filler

When there are areas in a record that you do not intend to reference in the output, you can use the input field name `filler`, one or more times, in order to account for all the record contents. The syntax is as follows:

```
/FIELD=(filler,POSITION=n,SIZE=n)
```
or
```
/FIELD=(filler,POSITION=n,SEPARATOR='char')
```

SEQUENCER

`SEQUENCER` is an internal field that keeps a running count of output records. The `SEQUENCER` field can be positioned and sized, as required, and is particularly useful when working with look-up tables and pseudonyms (see *Example 22* on page 69 for an example). It can be used one time in each `/OUTFILE` section of a **FieldShield** job script. The syntax is as follows:

```
/FIELD=(SEQUENCER[ = [+ / -]n], [field attributes])
```

where *n* is a whole number and you choose either "+" or "-" to indicate whether the initial value is positive or negative. If no initial value is given, then the initial value is 1. If no sign is given, then the sign is assumed to be positive.

Below are possible usages:

```
/FIELD=(SEQUENCER,POSITION=5,SIZE=4)
/FIELD=(SEQUENCER = -10,POSITION=5,SIZE=4)
/FIELD=(SEQUENCER = +100,POSITION=5,SIZE=4)
```

2.4 POSITION

This statement describes the starting location of each field in the record. The syntax is:

```
POSITION=n
or
POS=n
```

In a fixed-byte position field, *n* is the starting column for the first byte of the field. For example:

```
/FIELD=(lastname,POSITION=17,SIZE=10)
```

defines the lastname field as beginning at byte position 17 and having a width of 10 bytes.

In a variable-length (delimited) field, *n* is the field number when counting from left to right, with respect to the separator. For example:

```
/FIELD=(lastname,POSITION=2,SEPARATOR=',')
```

defines the lastname field as being the second field from the left within the record when the fields are separated by a comma (see SEPARATOR *on page 150*).

Position Over-Defines: Defining Multiple Fixed-Length Fields

You can over-define multiple fixed-position fields as a single `/FIELD` using the appropriate POSITION and SIZE attributes (see SIZE *on page 153*).

For example, considering the following input data:

```
George   Wallace     50000
George   Smith       51000
Henry    Jones       23000
```

You can combine multiple fields into one `/FIELD` statement, for example:

```
/FIELD=(firstname_lastname,POSITION=1,SIZE=16)
```

You can have fields defined for first name and last name, and you can define a field that contains both the first and the last name.

```
/FIELD=(firstname_lastname,POSITION=1,SIZE=16)

/FIELD=(firstname,POSITION=1,SIZE=7)
/FIELD=(lastname,POSITION=9,SIZE=7)
```

Position Over-Defines: Defining Contiguous Variable-Length Fields

You can over-define contiguous delimited fields as a single `/FIELD` using a variant of the `POSITION` attribute. Consider the following input data:

```
Stuart Clay|775-17-0363|01|CT
Taylor Guerrero|810-90-1269|02|MD
Charles Caldwell|906-43-3545|03|NY
Robyn Puckett|787-73-7773|04|NY
Santiago Lindsey|950-52-1240|05|TX
```

Note the above fields are name, ssno, IDnum, and state. The options for defining multiple variable-length fields are:

POSITION=:*n* From the beginning of the record to field *n*. For example:

```
/FIELD=(name_ssno_IDnum,POSITION=:3,SEPARATOR='|')
```

defines, as one field, the first three fields within the record.

POSITION=*n*: From field *n* to the end of the record. For example:

```
/FIELD=(ssno_to_end,POSITION=2:,SEPARATOR='|')
```

defines, as one field, the second field to the end of the record.

This is particularly useful when you have a large number of fields in a record, and you need to the map them to the output record, but do not intend to remap any individual fields within the multiple remaining fields of the record. In this way, you can over-define a single field to the end of a record rather than use multiple `/FIELD` statements to define every field that comprises the remainder of the record.

POSITION=*n1*:*n2* From field *n1* up to and including field *n2*. For example:

```
/FIELD=(ssno_IDnum,POSITION=2:3,SEPARATOR='|')
```

defines, as one field, the second and third fields within the record.

2.5 SEPARATOR

The separator character is used as the delimiting character that separates floating fields. The syntax is:

```
SEPARATOR=char[s]
```
or
```
SEP=char[s]
```

where `SEPARATOR` can be abbreviated to `SEP`, and where `char[s]` can consist of any of the following:

- a single ASCII character such as a comma (`,`)
- a multi-character ASCII string such as `,*|` (see Multi-Character Separators *on page 150*)
- a control character such as a tab (`\t`)
- a multi-byte character (see Multi-Byte Character Separators *on page 151*)
- hexadecimal character such as a # sign: (`\X23`) (see ASCII Collating Sequence *on page 239*).

NOTE

If your separator character is a single quote ('), you must escape it with a backslash character as follows:

```
SEPARATOR='\''
```

See Different Separators *on page 152* if you want to define fields with respect to different separator characters within the same record.

If you have records that use the comma (`,`) as the separator character, you might have records formatted as follows where you have two distinct fields:

```
Washington,George
Adams,John
Jefferson,Thomas
```

In this example, *field 1* could be called `lastname`, and *field 2* could be called `firstname`. In this case, your field definitions would be as follows:

```
/FIELD=(Lastname,POSITION=1,SEPARATOR=',')
/FIELD=(Firstname,POSITION=2,SEPARATOR=',')
```

Multi-Character Separators

You can also specify a multiple-character ASCII separator.

Example 46 Using a Multi-Character Separator

Consider the following data, **Produce2**:

```
Apples,*|Chicago,*|Red
Pears,*|Philadelphia,*|Yellow
Peaches,*|Macon,*|Peach
```

Use the following script, **multi_sep.fcl**, to define this input data and reformat it:

```
/INFILE=Produce2
   /FIELD=(Fruit, POSITION=1, SEPARATOR=",*|")
   /FIELD=(City, POSITION=2, SEPARATOR=",*|")
   /FIELD=(Color, POSITION=3, SEPARATOR=",*|")
/REPORT
/OUTFILE=stdout
   /FIELD=(Fruit, POSITION=1, SEPARATOR=",")
   /FIELD=(City, POSITION=2, SEPARATOR=",")
   /FIELD=(Color, POSITION=3, SEPARATOR=",")
```

On execution, the screen displays:

```
Apples,Chicago,Red
Pears,Philadelphia,Yellow
Peaches,Macon,Peach
```

NOTE

You can also incorporate control characters as part of a multi-character separator. For example, you can specify

```
SEPARATOR=',*\t'
```

to describe a separator that consists of a comma, an asterisk, and a tab.

Multi-Byte Character Separators

In cases where you are protecting fields with multi-byte characters (see Multi-Byte Data Types *on page 159*), you can specify a multi-byte separator character so that **FieldShield** can search for the separator as a whole character, rather than byte-by-byte.

NOTE

When using multi-byte data types, make sure that the last field in each record is terminated with the specified field separator. This is required by **FieldShield** because the record terminator may not be in the same encoding as the field data. Without the field terminator after the last field, **FieldShield** might not find the record terminator.

When the separator character is ASCII-compatible, such as UTF8 or SJIS, you can insert the character directly in the SEP attribute using your text editor, for example:

```
/FIELD=(last_name,POSITION=2,SEPARATOR='s',utf16)
```

where *s* is the ASCII-compatible separator character typed using the text editor.

For the Unicode data types UTF16 or UTF32, you must include the `%` character, the encoding type, and the UTF8 equivalent of the character within single quotes, for example:

```
/FIELD=(lname,POSITION=2,SEPARATOR=%utf16",",utf16)
```
or
```
/FIELD=(lname,POSITION=2,SEPARATOR=%utf32",",utf32)
```

NOTE

A multi-byte separator character is always required for records containing fields of data type UTF16 or UTF32.

See *Table 23* on page 219 for a list of **FieldShield**-supported multi-byte character data types.

Different Separators

FieldShield allows the use of multiple separators within the same record definition. The fields delimited by one separator are independent of the fields delimited by another.

Example 47 Using Different Separators

The **Produce1** file, used below, has two different separator characters:

- two fields delimited by a comma (,)
- three fields delimited by a pipe (|).

as shown here:

```
Apples|Chicago,IL|Red
Pears|Philadelphia,PA|Yellow
Peaches|Macon,GA|Peach
```

The following script file, **remap.fcl**, maps two fields to the output when there are two different separator characters defined in the input:

```
/INFILE=Produce1
    /FIELD=(Fruit, POSITION=1, SEPARATOR="|")
    /FIELD=(Address, POSITION=2, SEPARATOR="|")
    /FIELD=(State, POSITION=2, SEPARATOR=",", SIZE=2)
    /FIELD=(Color, POSITION=3, SEPARATOR="|")
/REPORT
/OUTFILE=stdout
    /FIELD=(State, POSITION=1, SEPARATOR="\t")
    /FIELD=(Fruit, POSITION=2, SEPARATOR="\t")
```

The Address field is given as `POSITION=2` because it is the second field in the record with respect to the pipe (|) separator. The State field is also specified at `POSITION=2` because it is the second field in the record with respect to the comma (,) separator. (See Multi-Byte Data Types *on page 159* for the exception to this convention.). The `SIZE` attribute is necessary here to ensure that only the first two characters are considered as the State.

On execution, the screen displays:

```
IL  Apples
PA  Pears
GA  Peaches
```

2.6 SIZE

This statement sets the width, in bytes, of a given field. The syntax is:

SIZE=*width*[.*precision*]

For example, to specify the `SKU` field as 13 bytes long, the statement could be:

```
/FIELD=(SKU,POSITION=16,SIZE=13)
```

Numeric Precision

The `SIZE=`*`width.precision`* option allows additional control of output data, for example, `SIZE=6.3`. When used with numeric data, *`.precision`* is used to indicate the number of digits required to the right of the decimal point, and *`width`* is the total length of the field, including the decimal point.

To demonstrate the use of numeric precision given a six-character input field, *Table 15* shows output based on size and type.

Table 15: Output Based on Size/Type

Input	Size	Data Type=NUMERIC
`123.45`	`8.2`	`_ _ 1 2 3 . 4 5`
`123.45`	`7.3`	`1 2 3 . 4 5 0`
`123.45`	`6.4`	`* * * * * *`
`abcdef`	`6.4`	`0 . 0 0 0 0`

`precision` may not exceed 18 bytes.

The `*` characters in the third row of *Table 15* signify that an overflow error has occurred in a numeric field. That is, the size specified is not sufficient to display the entire value of the field.

When working with delimited fields, it is recommended that you use the `PRECISION` option separately (see PRECISION *on page 154*) without a *`SIZE`* attribute, so that numeric field values of varying lengths need not conform to a fixed byte width.

2.7 PRECISION

To assign a uniform precision to numeric output values, without fixing the field size,
use the `PRECISION` option. Using `PRECISION` instead of the `SIZE=`*`width`*`[.`*`precision`*`]` convention (see SIZE *on page 153*) ensures that each numeric value is displayed with the precision you require. There will be neither wasted white space for smaller values, nor overflow for larger values. The syntax is:

```
PRECISION=n
```

where *`n`* is the numeric precision applied to a field.

Although supported for both fixed-length and delimited (variable-length) fields, use of `PRECISION` is recommended for delimited output fields. When you use the *`width.precision`* convention (see SIZE *on page 153*), all field widths will conform to the specified size, regardless of whether some values are shorter or longer.

For example, given these input records:

```
Harris,135,abc
Jones,5323.65,abc
Walters,923.0,abc
```

You can define the numeric field on output with only a `PRECISION` attribute:

```
/FIELD=(value,POSITION=2,PRECISION=2,SEPARATOR=',',NUMERIC)
```

and the following is returned:

```
Harris,135.00,abc
Jones,5323.65,abc
Walters,923.00,abc
```

Note that if you use the `SIZE` attribute instead, for example `/FIELD=(value,POSITION=2,SIZE=7.2,SEPARATRO=',',NUMERIC)`, the output is:

```
Harris, 135.00,abc
Jones,5323.65,abc
Walters, 923.00,abc
```

The numeric field has a width of 7 in each case, regardless of each input value's width (displaying white space where applicable).

2.8 FRAME

You can use the `FRAME` attribute for one or more fields. Typically, framed fields are used in a database table or Microsoft CSV data. Framing is useful in the following cases:

- when field contents contain a character that is also used as a delimiter, and you want to prevent **FieldShield** from treating such characters as field separators
- when you want to add, or change, a frame character from input to output.

The frame field attribute syntax is:

```
FRAME='char'
```

where *`char`* is the character that is used to frame the field.

Example 48 Using FRAME

Consider the following data, **framed_data**:

```
Grapes,"Washington D.C.",Black
Apples,"Chicago,IL",Red
Pears,"Philadelphia,PA",Yellow
```

This script, **frame.fcl**, can be used to isolate and remap the framed field (Address):

```
/INFILE=framed_data
    /FIELD=(FRUIT, POSITION=1, SEPARATOR=",")
    /FIELD=(ADDRESS, POSITION=2, SEPARATOR=",", FRAME='"')
    /FIELD=(COLOR, POSITION=3, SEPARATOR=",")
/REPORT
/OUTFILE=stdout
    /FIELD=(ADDRESS) # frame character removed
```

Note how the Color field is located at position 3 with respect to the separator. In the records where the address contains a comma (,) the field layout positions would be incorrect without using the `FRAME` attribute to define the Address field.

On execution, the screen displays:

```
Washington D.C
Chicago,IL
Philadelphia,PA
```

Note how commas within the framed field were not considered as a separator, and also, note how the frame character (") was removed on output because it was not specified in the output definition of that same field. The `"` or a different frame character could have been specified for the output.

NOTE

It is recommended that you define all framed input fields with the `FRAME` attribute. Otherwise, fields that may be referenced in your job that occur after undeclared framed fields might not be in their intended positions. This is because any delimiter characters existing in the undeclared frame fields will be wrongly considered as field separators.

2.9 Alignment

This field attribute is used when formatting ASCII fields. When used in the output section of a script, it aligns a desired field string (not its leading or trailing fill characters) to either the left or right of the field. Leading or trailing spaces are moved to the opposite side of the string. The following alignment options are accepted:

NONE_ALIGN	No change (the default).
LEFT_ALIGN	The string beginning with the first (non-space) character is aligned to the left of the target field. The remaining length to the right of the target field is populated with spaces or with a defined FILL character (see FILL *on page 158*).
RIGHT_ALIGN	Spaces to the right of the source string are removed. The remaining source string is moved to the right side of the target field. The remaining length to the left of the target field is filled with spaces or with a defined FILL character.

Example 49 Using RIGHT_ALIGN on Output

To right-align the president’s name field from the following input file:

```
McKinley William   1897-1901
Roosevelt Theodore 1901-1909
Taft William H.    1909-1913
Wilson Woodrow     1913-1921
Harding Warren G.  1921-1923
Coolidge Calvin    1923-1929
```

you would use a script like **right_align.fcl**:

```
/INFILE=chiefs_some
    /FIELD=(PRES, POSITION=1, SIZE=18)
    /FIELD=(TERM, POSITION=20, SIZE=9)
/REPORT
/OUTFILE=chiefs_some_right
    /FIELD=(PRES, POSITION=1, SIZE=18, RIGHT_ALIGN)
    /FIELD=(TERM, POSITION=20, SIZE=9)
```

The output is as follows:

```
  McKinley William 1897-1901
Roosevelt Theodore 1901-1909
   Taft William H. 1909-1913
    Wilson Woodrow 1913-1921
 Harding Warren G. 1921-1923
   Coolidge Calvin 1923-1929
```

The name string is right-aligned. The spaces existing within the strings have not been affected by the alignment.

Padding and Reducing

If you apply a left or right alignment attribute to the target field (output section of a job script), and assign it a length greater than on input, extra fill characters are inserted on the opposite side as required (see FILL *on page 158*). Conversely, fill characters are removed from the opposite side when a lesser length is given on output. If no length is specified for the target field, and output fields are delimited, then any fill characters are removed (not added to the opposite side). See *Trimming* below.

2.10 Trimming

If your output fields are delimited, you can trim all leading or trailing fill characters from a fixed- or variable-length input field using `LEFT_ALIGN` or `RIGHT_ALIGN`, and not specifying a length for the target field. In this case, any fill characters are trimmed and the non-fill characters are preserved.

Additional trim functionality is automatically included via the libcsutil library in `lib/modules/`. The syntax is:

`trim(`*`field`*`)` – Removes the leading and trailing white space.
`trim_left(`*`field`*`)` – Removes the leading white space.
`trim_right(`*`field`*`)` – Removes the trailing white space.

2.11 FILL

This statement is used for ASCII and non-binary numeric field statements. On input, it defines a pad character that must appear to the left of the field value. On output, it pads the left of the field with the character that you specify. If no FILL statement is used, the default fill character is a space.

Note that the default fill character for a non-binary field is a space; for a binary field, it is a binary null.

There are two forms of the `FILL` statement:

- `FILL='`*`char`*`'`
- `FILL=`*`n`*

where *`char`* is the fill character, and where *n* is the decimal weight of a character (see ASCII Collating Sequence *on page 239* for equivalents).

Notice the effect of the `FILL` statement in these examples:

```
/FIELD=(Value,POSITION=01,SIZE=07,NUMERIC)
/FIELD=(Value,POSITION=10,SIZE=07,FILL='0',NUMERIC)
/FIELD=(Value,POSITION=18,SIZE=11,FILL='*',CURRENCY)
```

If the input field `Value` is `12345`, the display would be:

```
  12345  0012345 $*12,345.00
||||||||||||||||||||||||||||
1234567890123456789012345678
0        1         2
```

2.12 Data Types (Single-Byte)

It is important to declare the data type of a field so that **FieldShield** can interpret the data correctly for conversions, and/or display purposes. For a description of all data types recognized in the **FieldShield**, see Data Types *on page 217*. If the data type is not given for an input or output field, the field is assumed to be ASCII.

NOTE

See Multi-Byte Data Types *on page 159* for details on using **FieldShield**-supported multi-byte data types.

Data-Type Conversion

When mapping from input to output, **FieldShield** can convert between data types. This can be another way to mask data or to improve the appearance of the output. For example, you can translate field data types from:

- numeric data to currency
- `EBCDIC` character to `ASCII`
- packed decimal values to binary integer

FieldShield also has the ability to convert multi-byte data types (see Conversion Between Multi-Byte Data Types *on page 160*).

2.13 Multi-Byte Data Types

FieldShield allows you to process, and convert between several multi-byte data types, including multiple Chinese, Korean, and Japanese collation standards.

See *Table 23* on page 219 and onward for a complete list of supported multi-byte data types.

If you specify one or more of the above data types, and you need to define multi-byte separator characters, see Multi-Byte Character Separators *on page 151*.

NOTE

When using multi-byte character separators, make sure that the last field in each record is terminated with the specified field separator. This is required by **FieldShield** because the record terminator may not be in the same encoding as the field data. Without the field terminator after the last field, **FieldShield** might not find the record terminator.

If you are specifying multi-byte data types, and you have more than one type of separator within each record, you cannot use the same `POSITION` value for multiple fields as you would with single-byte data types (as shown in Different Separators *on page 152*). Instead, you must use unique `POSITION`s for each subsequent field, whether the separator is the same or not.

For example, using the script from Different Separators *on page 152*, note how the `POSITION` values on input are unique when using multi-byte data types:

```
/INFILE=Produce
    /FIELD=(Fruit,POSITION=1,SEPARATOR=’|’,GBK)
    /FIELD=(Address,POSITION=2,SEPARATOR=’|’,GBK)
    /FIELD=(State,POSITION=3,SEPARATOR=’,’,GBK)
    /FIELD=(Color,POSITION=4,SEPARATOR=’|’,GBK)
/REPORT
/OUTFILE=stdout
    /FIELD=(State,POSITION=1,SEPARATOR=’|’,GBK)
    /FIELD=(Fruit,POSITION=2,SEPARATOR=’|’,GBK)
```

In the script above, note that each `POSITION` value in the input section is not necessarily the field’s position in the record with respect to the specified separator (as it is with Different Separators *on page 152*), but rather the value of `POSITION` is advanced with each newly defined field, regardless of whether the separator character changes (from | to , in this case).

Conversion Between Multi-Byte Data Types

FieldShield supports the following multi-byte data type conversions from input to output:

GBK on input - Converts to ASCII, UTF8, and UTF16.

SJIS on input - Converts to UTF8 and UTF16.

EHANGUL, HHANGUL, KEF on input - Converts to EUC_KR (WANSUNG).

GBK conversion to the ASCII data type will convert to its PINYIN (pronunciation) equivalent.

3 /INREC

The `/INREC` record provides a record definition to be used when **FieldShield** processes input source records. An /INREC is necessary when there are multiple input sources that do not have record definitions that are the same. It provides the opportunity to alter the format of the input before being passed to the output.

You can use /INREC to remove fields that are not necessary for further processing or to create new derived fields by processing input fields before doing the output processing. These can be done if there is one or more input sources.

In order to use /INREC when there are multiple input files with different record definitions, the input files must have fields that have the same name. You can then map and define these fields in /INREC. You may not map fields that are not common to all the input sources.

```
/INREC
   attributes
```

Example 50 Using /INREC

Consider the input file formats of the following files:

customerid:

```
SMITH,JOHN,12340987,CA
JOHNSON,MARY,98712345,AL
DOE,JANE,22223457,OH
```

transactions:

```
129.45,SHARON,JONES,89274629
345.00,MIKE,HARRIS,83957364
5847.98,JOSEPH,ALVAREZ,74635475
```

The following **FieldShield** script, **inrec.fcl**, describes their attributes, and maps a common record layout for them.

```
/INFILE=customerid
    /FIELD=(LNAME, POSITION=1, SEPARATOR=",")
    /FIELD=(FNAME, POSITION=2, SEPARATOR=",")
    /FIELD=(ACCTNUM, POSITION=3, SEPARATOR=",")
    /FIELD=(STATE, POSITION=4, SEPARATOR=",")
/INFILE=transactions
    /FIELD=(AMOUNT, POSITION=1, SEPARATOR=",")
    /FIELD=(LNAME, POSITION=2, SEPARATOR=",")
    /FIELD=(FNAME, POSITION=3, SEPARATOR=",")
    /FIELD=(ACCTNUM, POSITION=4, SEPARATOR=",")
/INREC
    /FIELD=(ACCTNUM, POSITION=1, SEPARATOR=",")
    /FIELD=(LNAME, POSITION=2, SEPARATOR=",")
    /FIELD=(FNAME, POSITION=3, SEPARATOR=",")
/REPORT
/OUTFILE=accounts
    /FIELD=(ENC_ACCTNUM=enc_aes256(ACCTNUM), POSITION=1, SEPARATOR=",")
    /FIELD=(LNAME, POSITION=2, SEPARATOR=",")
    /FIELD=(FNAME, POSITION=3, SEPARATOR=",")
```

The **acctnum** field has been encrypted in the output file, and the fields **lname** and **fname** have been mapped to the output file, **accounts**:

```
c4MUczRwyeNfheL94nc2cw==,SMITH,JOHN
T72GOyB0TGorFCeH3Fv7Og==,JOHNSON,MARY
PHb69C6UIc47aXUFT5c6QA==,DOE,JANE
wncblI2dLtOCM2EU4KibCg==,SHARON,JONES
Q5awTvzdpR6o/BHH3qkjhg==,MIKE,HARRIS
0wmvlMIF3ptghSZ023lRbw==,JOSEPH,ALVAREZ
```

4 Conditions

A condition is a *logical expression* that combines field names and/or constants with relational and/or logical operators. When the expression is evaluated, it will be either true or false.

A condition is associated with several **FieldShield** statements, and can be used for both input and output. Conditions allow you to assign contingencies to data protections and output formatting.

The true/false result determines how the statement works. The following statements use a condition within their definitions:

- `/INCLUDE and /OMIT`
- `/DATA using IF-THEN-ELSE logic`
- `/FIELD using IF-THEN-ELSE logic`

4.1 Syntax

A condition can be one of two forms (see Record Filtering (Omission) *on page 24*):

Named where the logical expression has a name and is defined with the statement:

```
/CONDITION=(condition_name, TEST=(logical_expression))
```

Once you have defined the `condition_name`, it may be used in one of the above statements that uses a condition, for example `/INCLUDE WHERE condition_name`. Naming is done for documentation purposes and to build nested expressions. It is also more efficient than using unnamed conditions if the same logical expression is used for more than one statement. The named condition is analyzed only once, and the result is reusable.

Unnamed (explicit) where the logical expression is built into the statement using it:

```
/INCLUDE WHERE logical_expression
/OMIT WHERE logical_expression
```

The `logical_expression` can involve any of the following types:

- Binary Logical Expressions
- Compound Logical Expressions *on page 164*
- You can use a C-style iscompare function in FieldShield to evaluate a condition based on the pattern matching expression that you specify. This is available at the field level (see Conditional Field Security and Masking on page 19), and also for record-filtering using `/INCLUDE` and `/OMIT` statements (see Record Filtering (Omission) on page 24). *on page 165*.

4.2 Binary Logical Expressions

Another form of a logical expression involves two values and a *relational operator*.
The following is the general form of the expression:

```
        Value1   Relational_Operator  Value2
```

The values can be field names or literals (constants), and `Value2` can contain a repetitive literal string such as `{180}"0"` to indicate 180 zeroes (see /Re-formatting and Masking Options *on page 74*). **FieldShield** recognizes both the operator and symbol forms of these relational operators, as shown in *Table 16*.

Table 16: Recognized Relational Operators

Operator	Symbol	Meaning
`EQ`	`==`	Equal
`NE`	`!=`	Not equal
`GT`	`>`	Greater than

`GE`	`>=`
	`!<`
		Greater than or equal
		Not less than

`LT`	`<`	Less than

`LE`	`<=`
	`!>`
		Less than or equal
		Not greater than

`CT`		Contains
`NC`		Does not contain

Examples of logical expressions using relational operators are:

```
Author EQ "Publisher"
Publisher >= "Addison-Wesley"
Price > 25.00
Value CT {10}"0"
"S" == $SOURCE
```

NOTE

The final example shows how you must reverse the standard compare order when an environment variable is used. Note also the difference between `Price > 25.00` (a numeric compare) and `Publisher >= "Addison-Wesley"` (a character compare). The CT example evaluates whether the `Value` field contains 10 zeroes.

4.3 Compound Logical Expressions

The simplest form of a compound logical expression is:

```
Expression 1  Logical_Operator  Expression 2
```

There are two logical operators:

AND	true when both Expression 1 *and* Expression 2 are true

OR true when either Expression 1 is true *or* Expression 2 is true.

An example of a compound logical expression is:

```
Publisher == "Dell" AND Price > 25.00
```

where `Publisher` and `Price` are previously defined fields.

4.4 Perl-Compatible Regular Expressions

You can use a C-style *iscompare* function in **FieldShield** to evaluate a condition based on the pattern matching expression that you specify. This is available at the field level (see Conditional Field Security and Masking *on page 19*), and also for record-filtering using `/INCLUDE` and `/OMIT` statements (see Record Filtering (Omission) *on page 24*).

NOTE

FieldShield takes machine locale into consideration when evaluating each compare (see /LOCALE *on page 182*):

The syntax is as follows:

ispattern(*field*,"*expression*"")
Checks the field using Perl-compatible regular expressions such as a+bc.

For example, consider the following input data, **names.in**:

```
Janet    20
James    10
Jane     30
Justin   25
Jack     35
Jeremy   40
Austin   50
```

The following uses a regular expression in an `/OMIT` statement to discard all names that begin with `Jan`:

```
/INFILE=names.in
   /FIELD=(name,POSITION=1,SIZE=8)
   /FIELD=(number,POSITION=9,SIZE=6)
/REPORT
/OUTFILE=names.out
   /OMIT WHERE ispattern(name,"[jJ][aA][nN][a-zA-z]+")
```

names.out contains:

```
James    10
Justin   25
Jack     35
Jeremy   40
Austin   50
```

4.5 Condition Evaluation Order

Any number of conditional expressions can be connected by logical operators, i.e.:

```
    Expr 1 Log Op 1 Expr 2 Log Op 2 Expr 3 ....
```

Evaluation proceeds left to right. For example, if `Expr 1` is TRUE and `Log Op 1` is OR, then the logical expression is TRUE. All evaluations are shown in *Table 17*.

Table 17: Evaluation of Logical Expressions

Unary/Binary Expression	Logical Operator	Result
True	None	True
False	None	False
True	AND	Continue
False	AND	False
True	OR	True
False	OR	Continue

4.6 Compound Conditions

It is possible to build conditions so that the logical expression of one condition will contain the name of a previously defined condition linked to the rest of the expression by a logical operator. Since parentheses are not recognized for grouping logical expressions, this is helpful in defining complex logical expressions. The following is an example of how to build these nested conditions:

```
/INFILE=chiefs
    /FIELD=(Name,POSITION=1,SIZE=27)
    /FIELD=(Party,POSITION=40,SIZE=3)
    /FIELD=(State,POSITION=45,SIZE=2)
    /CONDITION=(C1,TEST=(Party EQ "DEM")) # C1 defined
    /CONDITION=(C2,TEST=(Party EQ "REP")) # C2 defined
       /CONDITION=(C3,TEST=(C1 OR C2)) # C3 includes C1 and C2
    /OMIT WHERE C3                  # omit DEMs and REPs
/REPORT
/OUTFILE=out
```

You can build nested conditions so that the name of a previously defined condition is contained in the logical expression that defines another condition. This nesting is used instead of grouping logical expressions within parenthesis.

When defining a `WHERE` clause, you cannot use AND or OR logic that references multiple condition names. In the above script, for example, you can not have:

```
/OMIT WHERE C1 OR C2
```

Instead, you must first define a new, single condition that contains the compound condition logic you require, such as C3 in the above example, and then reference that single name where required.

Alternatively, you can use explicit compound condition logic without pre-defining conditioning names earlier in the script, for example:

```
/OMIT WHERE Party EQ "DEM" OR Party EQ "REP"
```

5 Data Source and Target Formats (/PROCESS)

The structure of an input or output file is described using the `/PROCESS` statement. This statement is used to identify those special file structures that **FieldShield** can process as an input source, or produce as a target. You can also convert from one file structure to another, that is, you can specify one process type on input, and a different process type on output. If no process is specified for either the input or output, the default file structure is `RECORD`.

The syntax for a `/PROCESS` declaration is:

/PROCESS=*process_type*

FieldShield recognizes the following file types:

- RECORD or RECORD_SEQUENTIAL *on page 168*
- MFVL_SMALL *on page 169*
- MFVL_LARGE *on page 170*
- VARIABLE_SEQUENTIAL (or VS) *on page 170*
- LINE_SEQUENTIAL (or LS) *on page 170*
- VISION *on page 171*
- VSAM *on page 171*
- MFISAM *on page 171*
- UNIVBF *on page 171*
- CSV *on page 172*
- LDIF *on page 172*
- ODBC *on page 173*
- BLOCKED *on page 180*
- XML *on page 180*
- ELF (W3C Extended Log Format) *on page 181*.

5.1 RECORD or RECORD_SEQUENTIAL

If no process is given, or the process type is explicitly `RECORD`, every input source or output target is a text file of either fixed or variable length:

fixed A length statement can be given when all records are of the same length. This is given in the following form:

```
/LENGTH=n
```

where every record is length *n*, which can be a maximum of 65,535 bytes. In COBOL, this is equivalent to `RECORD_SEQUENTIAL`.

If your input is linefeed-terminated, but you are certain that all input records are of equal length, then in some cases, you can improve job efficiency by specifying a fixed-length for the input file. Be sure to add one or two bytes to the record length to accommodate each record’s LF or CRLF, respectively. If the entire record is to be output, then specify the same length on output as you did on input to avoid having double linefeeds after each output record.

If you declare a fixed length as an attribute for an input file, and you do not specify a field layout for its respective output file, you must include the same `LENGTH=n` statement from the input file as an attribute for the output file.

NOTE

The process type `RECORD_SEQUENTIAL` or `RS` is an alias of the process type `RECORD`.

If one or more records in the input data is not the record length you have declared, your output results may be offset.

If the last record in the input data is shorter than the record length you have declared, this last record is discarded. This can happen if your last record is either a tail record, or contains only the excess characters of a previous record (in the event of offset data).

It is therefore suggested that you declare records as fixed-length only when you are certain that every record in your input data is of equal length. To determine this, divide the file size by the record length. If the result is not an whole number, then either the record length declaration is incorrect, or there is a wrong-sized record within your input data.

variable If no `/LENGTH` is specified, or you specify `/LENGTH=0`, then records may vary from 0 to 65,535 bytes in length and terminate on a line-feed character. This is equivalent to line sequential where there are no data byte values between x00 and x1F.

NOTE

Depending on the operating system, the length of a variable-length record is the position number of the last character of the last field on the line, plus one for the line-feed character, or plus two bytes for carriage-return and line-feed.

When processing variable-length records with binary values, your data must not have a binary 10 (hex 0a), the line-feed character, before the end of the record. The value would be taken as the record terminator.

5.2 MFVL_SMALL

This describes Micro Focus variable-length records. The file has a 128-byte header record, and each record is prepended with a binary short integer of two bytes that holds the size of the record. The maximum record length is 4,096 bytes. Each record begins at an offset address that is evenly divisible by four.

When you are using this process type on output, you have the option to set your own minimum and maximum record lengths, and the values will be written to the header record to support inter-process conversions.

The syntax for using this process on output therefore supports additional options:

```
/PROCESS=MFVL_LARGE[,min_length][,max_length]
```

where `min_length` and `max_length` are the minimum and maximum record lengths contained in the output file. If `/PROCESS=MFVL_SMALL` on input, this information is taken from the input file if you do not specify these attributes.

5.3 MFVL_LARGE

This describes Micro Focus variable-length records. In most cases, the file has a 128-byte header record, and each record is prepended with a binary int of four bytes that holds the size of the record. The maximum record length is 268,435,455 bytes. Each record begins at an offset address that is evenly divisible by four.

When you are using this process type on output, you have the option to set your own minimum and maximum record lengths, and the values will be written to the header record to support inter-process conversions.

The syntax for using this process on output therefore supports additional options:

```
/PROCESS=MFVL_LARGE[,min_length][,max_length]
```

where `min_length` and `max_length` are the minimum and maximum record lengths contained in the output file. If `/PROCESS=MFVL_LARGE` on input, this information is taken from the input file if you do not specify these attributes.

5.4 VARIABLE_SEQUENTIAL (or VS)

This describes variable-length records that are a string of characters prepended by a short integer. The value of the short is the length of the record. The hex representation of the short is machine dependent. Most unix and mainframe computers use the big endian representation. When migrating this type of file between computers that have a different endianness, you must convert the short into the endianness of the destination computer.

5.5 LINE_SEQUENTIAL (or LS)

This describes variable-length records that are strings of characters where each low value data byte (x00 through x1f) is *protected* (or prepended) by a null byte (x00).
The terminating linefeed character is not protected.

5.6 VISION

This describes ACUCOBOL-GT Vision files, which is a proprietary index file format from Micro Focus.

NOTE

To obtain the requisite Vision-enabled **FieldShield** libraries for linking to Vision libraries, contact your IRI agent.

5.7 VSAM

This describes records in Clerity's proprietary VSAM format. When available with Clerity's Mainframe Batch Manager (MBM) and Mainframe Transaction Processing (MTP) software, **FieldShield** is linked to their C-ISAM library. VSAM records can then be moved from, and to, **FieldShield** for processing.

NOTE

`/PROCESS=VSAM` is not supported on Windows.

To obtain the requisite VSAM-enabled **FieldShield** library for linking to your VSAM libraries, contact your IRI agent.

5.8 MFISAM

This describes records in Micro Focus indexed file format (MF ISAM).

NOTE

To obtain the requisite MF ISAM-enabled **FieldShield** libraries for linking to your MF ISAM libraries, contact your IRI agent.

5.9 UNIVBF

This describes variable-length records that are prepended by a 4-byte ASCII field giving the length of the record, including the four bytes. This format may be generated by Unisys mainframes writing to tapes.

5.10 CSV

Specifying `/PROCESS=CSV` instructs **FieldShield** to recognize the data file as a Microsoft **.csv** (comma-separated values) file. CSV files always contain ASCII fields separated by commas, and a header that names the file’s fields in order from left to right. CSV data can contain commas within individual fields if such fields are enclosed in double quotes. **FieldShield** will skip a file’s header when `/PROCESS=CSV` is specified. To force **FieldShield** to generate a header based on the file’s field names and positions, specify `/PROCESS=CSV` in the output file. See Example *on page 197* for details on using `/PROCESS=CSV`.

NOTE

The maximum header length supported by the use of `/PROCESS=CSV` is 4,096. If the header length exceeds 4,096 bytes, you can use the `/HEADSKIP` statement (see /HEADSKIP *on page 28*).

For the purposes of **FieldShield**, it is expected that the first record of a Microsoft **.csv** file data is a header record. Therefore, if the input file does not have a header, you cannot use the `/PROCESS=CSV` statement.

To write specifications for processing CSV data files, you may wish to use the utility **csv2ddf**. For Windows users, this utility is provided in the **`\`*`install_dir`*`\bin`** directory. For UNIX and Linux users, **csv2ddf** is provided in the **$FIELDSHIELD_HOME/bin** directory. (See csv2ddf *on page 196*.) The utility examines file headers and generates a **FieldShield** data definition file based on the input field names found in the CSV file header.

5.11 LDIF

Use `/PROCESS=LDIF` for files in LDIF (Lightweight Directory Interchange) format (that is, the format of data exported from an LDAP database). In the input section of a **FieldShield** job script, you must define each desired LDIF attribute as a `/FIELD` statement, and if a value does not exist for a specific attribute, the value will be null on output. When using `/PROCESS=LDIF`, data columns must be defined as delimited fields, that is, you must specify a separator character (see SEPARATOR *on page 150*).

On output, when using `/PROCESS=LDIF,` the attribute name will be the same as the field name. If the field value is null, the attribute will not appear in the output target. You do not have to specify a separator character in any of the fields when LDIF is the outfile process.

To write specifications for processing LDIF files, you may wish to use the utility **ldif2ddf**. For Windows users, this utility is provided in the **`\`*`install_dir`*`\bin`** directory. For UNIX and Linux users, **ldif2ddf** is provided in the **$FIELDSHIELD_HOME/bin** directory. See ldif2ddf *on page 203*.

5.12 ODBC

You can use `/PROCESS=ODBC` in a job script to process (on input) and populate (on output) table columns in databases supported by ODBC (Open Database Connectivity). For **FieldShield** version 1.5, the following database types have been tested for compliance with `/PROCESS=ODBC`:

- Oracle
- SQL Server
- DB2
- MySQL

NOTE

On some systems, the ODBC specification has been updated such that a certain value (SQLLEN) has changed from a 32-bit integer to a 64-bit integer when the software is compiled in 64-bit mode. Some older drivers will expect this value to be 32-bit, and newer drivers will expect it to be 64-bit. To address this, a separate file format must be used, `/PROCESS=ODBC_LEGACY`, which supports the older standard. Current PostgreSQL and MySQL ODBC drivers use the 64-bit value (`/PROCESS=ODBC`). Oracle 11i ODBC drivers use the 32-bit value (`/PROCESS=ODBC_LEGACY`). This is not an issue on Windows or MacOSX (iODBC always defines it as 64-bit).

W Requirements for Windows Users Using /PROCESS=ODBC

In order to use `/PROCESS=ODBC` or **odbc2ddf** (see odbc2ddf *on page 207*), you must first create a Database Source Name (DSN) for each source or target you intend to connect to. Connection requires the presence of an ODBC driver. The ODBC driver must be installed on the specific machine or server on which **FieldShield** resides. Windows machines typically contain an ODBC manager called the ODBC DATA Source Administrator that manages database drivers and data sources, and maintains a list of DSNs. On 64-bit Windows systems, there is a native 64-bit ODBC manager that also offers a 32-bit ODBC manager. If you have a 32-bit **FieldShield**, you must create DSNs using a 32-bit ODBC manager. If you have a 64-bit operating system while running a 32-bit **FieldShield**, you must create DSNs using a 32-bit ODBC Data Source Administrator. ◆

U Requirements for UNIX/Linux Users Using /PROCESS=ODBC

In order to use `/PROCESS=ODBC` or **odbc2ddf** (see odbc2ddf *on page 207*), you must manually set up the ODBCUnix driver manager on your system. The version of `odbcunix` must be 2.2.14 or higher, and it must match the bit architecture of your **FieldShield** version (32- or 64-bit). You must also have an ODBC data source driver (for each database your are connecting with) that matches the bit architecture of **FieldShield**. Using the ODBC driver manager, you must create a Database Source Name (DSN) for each source or target you intend to connect to. ◆

5.12.1 Syntax: Using /PROCESS=ODBC

For all job scripts that contain /PROCESS=ODBC, you must move or copy the library file **libcsodbc.so** (UNIX/Linux) or **libcsodbc.dll** (Windows) from **/lib** in the home directory to **/lib/modules**. If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The file can also be referenced from the current working directory.

When using `/PROCESS=ODBC` to extract and process a database table, or to specify a target table to be populated, you must indicate the table name and the Database Source Name (DSN) in the `/INFILE` or `/OUTFILE` statement, respectively, and enclose these details in quotes, for example:

```
/INFILE="tablea;DSN=oracledb[;UID=username;PWD=password;]"
    /PROCESS=ODBC
```

or

```
/OUTFILE="tableb;DSN=oracledb[;UID=username;PWD=password;]"
    /PROCESS=ODBC
```

where `tablea` is the source target to be extracted, and `tableb` is the target table to be populated. `oracledb` is the DSN. The `UID` and `PWD` are dependent on the system odbc configuration. If they are specified in the system ODBC configuration, they are not required in the `/INFILE` or `/OUTFILE` statement.

All data columns must be defined as delimited fields, as in the following example where `/PROCESS=ODBC` is used to identify both a database source and a database target:

```
/INFILE="tablea;DSN=oracle_tester;UID=scott;PWD=tiger;"
    /PROCESS=ODBC
    /FIELD=(field1,POSITION=1,EXT_FIELD="field1",SEPARATOR=',')
    /FIELD=(field2,POSITION=2,EXT_FIELD="field2",SEPARATOR=',')
    /FIELD=(field3,POSITION=3,EXT_FIELD="field3",SEPARATOR=',')
/REPORT
/OUTFILE="tableb;DSN=oracle_tester;UID=scott;PWD=tiger;"
    /PROCESS=ODBC
    /FIELD=(field1,POSITION=1,EXT_FIELD="field1",SEPARATOR=',')
    /FIELD=(field2,POSITION=2,EXT_FIELD="field2",SEPARATOR=',')
    /FIELD=(field3,POSITION=3,EXT_FIELD="field3",SEPARATOR=',')
```

Note that you should choose a separator character that will not appear in any of the data values. Common choices are comma (,), tab (\t), and vertical pipe(|).

5.12.2 EXT_FIELD

In some cases, you may want to specify a `/FIELD` name that is specifically meaningful in the context of that job script. The desired field name will differ from its corresponding column name in the database. In this case, you may use the field attribute `EXT_FIELD` to assign the database column name to the field (to ensure that the field data will be extracted and loaded properly when using `/PROCESS=ODBC` on input and output, respectively), while using the desired **fieldshield** field name that you choose.

For example, if you are extracting data from a table with the column name SS_NO, and prefer to refer to this column as ssn in **fieldshield**, use the following for example:

```
/FIELD=(ssn,EXT_FIELD="SS_NO",POSITION=1,SEPARATOR=',')
```

NOTE

If your **fieldshield** `/FIELD` name differs from its corresponding name in the database, the field data will not be extracted or loaded and an error will occur. You must use `/EXT_FIELD` to prevent this.

To write specifications for processing ODBC data files, you may wish to use the utility **odbc2ddf**, which is provided with the **FieldShield** package in the **$FIELDSHIELD_HOME/bin** directory. In Windows, **odbc2ddf** is located in `\`***`install_dir`***`\bin` (see odbc2ddf *on page 207*). The utility examines ODBC-supported database tables and generates a **fieldshield** data definition file based on the detected layout.

5.12.3 Extraction

In the input section of a **fieldshield** script, you must define each table column you want to extract using corresponding `/FIELD` statements for those columns. See odbc2ddf *on page 207* for details on using **odbc2ddf**, which will automatically provide a complete list of `/FIELD` statements based on the column descriptions from the database. Several ODBC data types have **fieldshield** data type equivalents, as shown in *Table 19* on page 209, so you must specify `/FIELD` statements for only those database columns whose data types have **fieldshield** equivalents.

QUERY

The /QUERY= statement is a string that contains a valid SQL SELECT statement. All of the columns that are returned from the query are mapped to the declared fields in the section by position, not by field name. The field or column names are ignored.

When using PROCESS=ODBC, fieldshield creates a query statement to process information in the script. You can create a customized query statement that overrides this internal query statement. The following is an example of a valid query:

```
/INFILE="SCOTT.CHIEFS;DSN=oratester"
   /PROCESS=ODBC
   /ALIAS=SCOTT_CHIEFS
   /QUERY="SELECT LNAME,FNAME,TERM,PARTY FROM SCOTT.CHIEFS WHERE PARTY='DEM'"
   /FIELD=(LNAME, TYPE=ASCII, POSITION=1, EXT_FIELD="LNAME", SEPARATOR="|")
   /FIELD=(FNAME, TYPE=ASCII, POSITION=2, EXT_FIELD="FNAME", SEPARATOR="|")
   /FIELD=(TERM, TYPE=ASCII, POSITION=3, EXT_FIELD="TERM", SEPARATOR="|")
   /FIELD=(PARTY, TYPE=ASCII, POSITION=4, EXT_FIELD="TERM", SEPARATOR="|")
/REPORT
/OUTFILE="SCOTT.CHIEFS;DSN=oratester"
   /PROCESS=ODBC
   /ALIAS=SCOTT_CHIEFS
   /FIELD=(LNAME, TYPE=ASCII, POSITION=1, EXT_FIELD="LNAME", SEPARATOR="|")
   /FIELD=(FNAME, TYPE=ASCII, POSITION=2, EXT_FIELD="FNAME", SEPARATOR="|")
   /FIELD=(TERM, TYPE=ASCII, POSITION=3, EXT_FIELD="TERM", SEPARATOR="|")
   /FIELD=(PARTY, TYPE=ASCII, POSITION=4, EXT_FIELD="TERM", SEPARATOR="|")
```

NOTE

Fieldshield is designed to process structured, flattened data. Errors will occur if carriage returns, linefeeds, frame characters, etc. are detected within data values when processing database tables.

5.12.4 Loading

When using `PROCESS=ODBC` on output, the output `/FIELD` statements must contain data types that have ODBC equivalents, as shown in *Table 19* on page 209. You can populate only those tables that have already been created in the database, that is,
`/PROCESS=ODBC` does not generate a *create table* statement.

When database tables are populated using `/PROCESS=ODBC` on output, tables will be populated using an *append* (`/APPEND`) option by default, where existing rows of table data will remain intact, and new table rows will be appended. You can change this default behavior by using either of the following statements: `/CREATE` or `/UPDATE`.

/APPEND

Associate an `/APPEND` with any target to cause output data to be placed after the existing data in a file or table. If the file does not exist or is empty, `/CREATE` will be invoked.

The syntax is:

```
/OUTFILE=output_filename
/APPEND
```

/CREATE

This is the default specification for an output file. It indicates that a new output file will be created or an existing table will be truncated. If the file name already exists, all previous data in the file will be lost, even if nothing is written by this job.

The syntax is:

```
/OUTFILE=output target
/CREATE
```

/UPDATE

The /`UPDATE` statement introduces a third option to the choice of /`APPEND` and /`CREATE`. The /`UPDATE` command must include one or more key fields to use in the WHERE clause of the resulting UPDATE command in SQL. Note that the key fields must exist somewhere in the INPUT.

The syntax is:

```
/OUTFILE=output_filename
/PROCESS=ODBC
/UPDATE=(field)
```

Example Using Update

In the following example, the `IDNUMBER` is not being changed, but is needed for the key only, which must be unique. This is usually the primary key.

```
/INFILE="PATIENT_RECORD;DSN=Oracle"
   /PROCESS=ODBC
   /FIELD=(IDNUMBER, ASCII,POSITION=1,SEPARATOR="\t")
   /FIELD=(CREDIT_CARD,ASCII,POSITION=2,SEPARATOR="\t")
   /FIELD=(DRIVING_LICENSE,ASCII,POSITION=3,SEPARATOR="\t")
/REPORT
/OUTFILE="PATIENT_RECORD;DSN=Oracle"
   /PROCESS=ODBC
   /UPDATE=(IDNUMBER)
   /FIELD=(ENC_FP_CREDIT_CARD=enc_fp_aes256_alphanum(CREDIT_CARD,"file:pass-
word"),\
       ASCII,POSITION=1,SEPARATOR="\t",EXT_FIELD="CREDIT_CARD")
   /FIELD=(ENC_FP_DRIVING_LICENSE=enc_fp_aes256_alphanum(DRIVING_LICENSE,\
      "file:password"),ASCII,POSITION=2,SEPARA-
TOR="\t",EXT_FIELD="DRIVING_LICENSE")
```

In the outfile, note that only two fields, `CREDIT,CARD` and `DRIVING_LICENSE`, are updated and encrypted, based on the `IDNUMBER` key.

Using the same infile and action as above, the following example updates and encrypts all three fields: `IDNUMBER`, `CREDIT_CARD`, and `DRIVING_LICENSE`. Note that when the key is also PII and needs to be changed (across the entire database to maintain integrity), you could change it to a new value, based on matching its old value.

```
/INFILE="PATIENT_RECORD;DSN=Oracle"
   /PROCESS=ODBC
   /FIELD=(IDNUMBER, ASCII,POSITION=1,SEPARATOR="\t")
   /FIELD=(CREDIT_CARD,ASCII,POSITION=2,SEPARATOR="\t")
   /FIELD=(DRIVING_LICENSE,ASCII,POSITION=3,SEPARATOR="\t")
/REPORT
/OUTFILE="PATIENT_RECORD;DSN=Oracle"
   /PROCESS=ODBC
   /UPDATE=(IDNUMBER)
   /FIELD=(ENC_FP_IDNUMBER=enc_fp_aes256_alphanum(IDNUMBER, "file:password"), \
        ASCII,POSITION=1,SEPARATOR="\t",EXT_FIELD="IDNUMBER")
   /FIELD=(ENC_FP_CREDIT_CARD=enc_fp_aes256_alphanum(CREDIT_CARD,"file:pass-
word"),\
       ASCII,POSITION=2,SEPARATOR="\t",EXT_FIELD="CREDIT_CARD")
   /FIELD=(ENC_FP_DRIVING_LICENSE=enc_fp_aes256_alphanum(DRIVING_LICENSE,\
      "file:password"),ASCII,POSITION=3,SEPARA-
TOR="\t",EXT_FIELD="DRIVING_LICENSE")
```

NOTE

If you have large data, and your load time speeds are not satisfactory using `/PROCESS=ODBC` on output, it is recommended that you instead use the bulk load facility supported by your target database.

For this purpose, tab-delimited `/FIELD`s are generally supported (`SEPARATOR='\t'`), with `/PROCESS=RECORD` in the output section, rather than `/PROCESS=ODBC`. You can then load the **fieldshield** output into your database table with the bulk load facility.

5.12.5 Special Considerations

Binary numeric data type columns from the database, such integer and double, must also be specified as delimited fields in **fieldshield**, and given a data type of `NUMERIC`. In the case of ODBC data types INTEGER and SHORT with no decimal places, you must specify `NUMERIC` with `PRECISION=0` in **fieldshield** (see PRECISION *on page 154*).Binary floating point types such as DOUBLE should be specified as `NUMERIC` (though precision can be set optionally). If you specify any of these field types as ASCII, **fieldshield** can not perform any mathematical operations on the field, however you can extract and load such data in the database when defined as ASCII.

Regarding database date and timestamp columns, depending on how you specify the field in **fieldshield** on the input side, `/PROCESS=ODBC` will ensure that the column data is converted into that specific format, for example ISO_DATE will convert the database column to a format like `2006-09-19` (see *Table 35* on page 234 for the format of date and timestamp data types). These specifications are made in field statements on the input side because using `/PROCESS=ODBC` retrieves date and timestamp values in a binary format, and converts them to whichever data type or string format is required. If using `/PROCESS=ODBC` on output, you should not perform any data type conversion on these date and timestamp fields. Such columns will load into the target correctly, regardless of the declaration you used on the input side.

All other database data types should be specified as ASCII. *Table 19* on page 209 lists the ODBC data types that have data type equivalents.

5.13 BLOCKED

The blocked file format is employed by various tape drives, and describes variable-length records that are enclosed in variable-length blocks of records.

The 4-byte block-size indicator is encountered first by **FieldShield**. It is then followed by a stream of 4-byte record-size indicators, followed by record data. When the accumulated records reach the current block size, a new block is introduced with its size indicator, and a new record stream follows.

NOTE

`/PROCESS=BLOCKED` is not supported on output.

5.14 XML

When `/PROCESS=XML` on input, flat-file records formatted with XML tags are extracted and processed by **FieldShield**. When `/PROCESS=XML` on output, records are generated within XML tags. The XML attribute and tag names, for both **FieldShield** input and output purposes, are specified in an XDEF attribute for each `/FIELD` statement to define the XML data elements. Alternatively, for either input or output purposes, you can specify a single XDEF for one field, and the same specified nodes will be applied to all remaining fields, where the tag names will assume the `/FIELD` names by default (either the input or output `/FIELD` names for the input or output XML process, respectively).

To write specifications for processing XML data files, you may wish to use the utility **xml2ddf**. For Windows users, this utility is provided in the **`\`*`install_dir`*`\bin`** directory. For UNIX and Linux users, **xml2ddf** is provided in the **$FIELDSHIELD_HOME/bin** directory. (See xml2ddf *on page 205*.)

If you do not define any XDEF attributes when /PROCESS=XML on output, the XML tag definitions will default to a generic /File/Record naming convention in the target XML file. However, if the input section contains its own XDEF attributes that define an XML source, these FIELD statements and XDEFs will map automatically to output, and be applied to the target, unless you explicitly specify output fields.

FieldShield supports only those XML files whose data elements conform to a flattened structure. FieldShield can extract one element of the same name at a given level. If you have multiple tags of the same name, FieldShield will extract the last occurring tag of a given name.

When defining the XDEF attribute, the syntax for `/FIELD` statements is:

```
/FIELD=(fieldname,POSITION=n,SEPARATOR='x',XDEF="/node1/node2/[/node3]/tag_name")
```

Optionally, you can use the `@` sign to specify that data is an attribute of the XML tag preceding it, for example:

```
/FIELD=(fieldname,POSITION=n,SEPARATOR='x',XDEF="/node1/node2/@attribute_name")
```

Note that you can specify any level of *node*s, depending on how nested the source XML tag is for input purposes, or how nested you would like the XML target tag to be for output purposes.

NOTE Generally, the first two nodes listed in an XDEF attribute will be the same for all fields you are defining because the actual data elements do not typically reside within the two uppermost node levels in the XML hierarchical structure.

The `SIZE` and `POSITION` (or the `SEPARATOR` and `POSITION`) attributes are required for XML input fields defined in **FieldShield**, but they have only internal significance. When using `/PROCESS=XML` on input, it is therefore recommended that you specify delimited input fields in order to accommodate field values of any size within the source file
(see POSITION *on page 148* and SEPARATOR *on page 150*).

The `/FIELD` statements that you define on output will determine the size and position of the field elements within the records that are output by **FieldShield**. However, if no output fields are explicitly specified, note that the size and position attributes from the input section will be applied automatically to the output target.

5.15 Web Logs

ELF (W3C Extended Log Format)

FieldShield supports internet web transaction files in W3C Extended Log Format (ELF). These files have a header containing lines of comments, followed by a line naming the data fields. To instruct **FieldShield** to skip the header, specify

`/PROCESS=ELF` in the input file. To force **FieldShield** to generate a header based on the file's field names and positions, specify `/PROCESS=ELF` in the output file.

If you wish to write specifications for processing ELF data files, you can use the **FieldShield** utility **elf2ddf**. For Windows users, this utility is provided in the **`\`*`install_dir`*`\bin`** directory. For UNIX and Linux users, **elf2ddf** is provided in the **$FIELDSHIELD_HOME/bin** directory. (See elf2ddf *on page 200*.)
This utility reads ELF file headers and generates **FieldShield** data definition files accordingly, giving users a base for writing specifications. Its syntax is:

```
elf2ddf datafile [data-definition-filename]
```

CLF (NCSA Common Log Format)

Web logs in CLF format can also be processed within **FieldShield**. Example data definition files **CLF_Referrer.ddf**, **CLF_Agent.ddf**, and **CLF_Access.ddf** are provided in the directory **\*install_dir*\examples\fieldshield** on Windows, and in the **$FIELDSHIELD_HOME/examples/fieldshield** directory on UNIX and Linux.

6 /LOCALE

FieldShield allows the user to set another language environment. This will affect the following processes:

- character classification and case conversion
- numeric formatting
- some message language
- currency formatting
- date and time formatting.

To set the **FieldShield** locale, use the following command:

```
/LOCALE=[language[_territory[.codeset]]]
```

For the available locale options on various operating systems, you can refer to your system's International Language Support documentation. If your operating system does not support your desired language, contact IRI for further assistance.

If the `/LOCALE` command is used without options, the locale is the current environment. Therefore, the `/LOCALE` command by itself is needed when switching back to the current locale.

Example 51 Using /LOCALE with Currency

The values in the input file **values.dat** will be converted to British currency:

```
5
16
138
0
227
```

The following file, **ukvalues.fcl**, uses a `/LOCALE` command (with language option) to perform the conversion:

```
/LOCALE=en_UK   # sets the language
/INFILE=values.dat
    /FIELD=(value, POSITION=1, SIZE=3, NUMERIC)
/OUTFILE=stdout
    /FIELD=(value, POSITION=1, SIZE=8, CURRENCY)
                # resized for extra characters
```

On execution, **ukvalues.out** contains:

```
  £5.00
 £16.00
£138.00
  £0.00
£227.00
```

The above example was performed on a Solaris operating system. *Table 18* on page 183 shows all supported system-specific `/LOCALE` options.

NOTE Depending on your system, not all locale options may be installed.

Table 18: System-Specific Locale Options

Language in Territory	AIX	HP-UX[a]	Solari
Default "C "locale		C - default	C
Arabic		arabic arabic.iso88596 arabic-w	ar
Basque			eu
Bulgarian		bulgarian	bg

Table 18: System-Specific Locale Options (cont.)

Language in Territory	AIX	HP-UX[a]	Solari
Canadian French	Fr_CA fr_CA	c-french c-french.iso88591	fr_CA
Catalan			ca
Chinese		chinese-s	zh
Chinese in Taiwan	zh_TW	chinese-t chinese-t.big5	zh_T W
Corsican			co
Czech		czech	cs
Danish	Da_DK da_DK	danish danish.iso88591	da
Dutch	NL_NL nl_NL	dutch dutch.iso88591	nl
Dutch in Belgium	NL_BE nl_BE		
English			en
English in UK	En_GB en_GB	english english.iso88591	en_UK
English in US	En_US en_US	american american.iso88591	en_US
Esperanto			eo
Finnish	Fi_FI fi_FI	finnish finnish.iso88591	fi
French in Belgium	Fr_BE fr_BE		fr_BE
French in France	Fr_FR fr_FR	french french.iso88591	fr
Frisian			fy
German	De_DE de_DE	german german.iso88591	de
Greek	el_GR	greek greek.iso88597	el
Greenlandic			kl
Hebrew		hebrew hebrew.iso88598	iw
Hungarian		hungarian	hu
Icelandic	Is_IS is_IS	icelandic icelandic.iso88591	is
Irish			ga
Italian	It_IT it_IT	italian italian.iso88591	it

Table 18: System-Specific Locale Options (cont.)

Language in Territory	AIX	HP-UX[a]	Solari
Japanese	Ja_JP ja_JP	japanese japanese.euc katakana	ja
Korean	ko_KR	korean	ko
Latvian			lv
Norwegian	No_NO no_NO	norwegian norwe-	no
Persian			fa
Polish		polish	pl
Portuguese	Pt_PT pt_PT	portuguese portu-	pt
Romanian		rumanian	ro
Russian		russian	ru
Scots Gaelic			gd
Serbian			sr
Serbo-Croatian		serbocroatian	sh
Slovak		slovene	sk
Spanish	Es_ES es_ES	spanish spanish.iso88591	es
Swedish	Sv_SE sv_SE	swedish swedish.iso88591	sv
Swiss French	Fr_CH fr_CH		fr_CH
Swiss German	De_CH de_CH		de_CH
Thai		thai	
Turkish	tr_TR	turkish turkish.iso88599	tr
Welsh			cy
Yiddish			ji

a. iso88591 is a Latin-1 character codeset, an extended ASCII standard that uses all eight bits of a byte, filling in the top half of the character set with accent marks and special characters used in Western Europe.

The command `locale -a` lists all the locales available for use. For a list of Linux locale options, see `http://wiki.debian.org/LocaleSupported`.

Custom Procedures

1 Custom Input

In addition to the `/INFILE` option supported by **FieldShield** (see Input Files *on page 139*), you can also write and invoke your own custom input procedure. **FieldShield** allows the calling program to set up a custom input procedure by way of the command `/INPROCEDURE`.

You can use a custom input procedure to process flat records from a proprietary DBMS or ETL tool, for example. In these cases, you can program a *hook* between the API of the tool (such as ODBC or OCI) and your **FieldShield** input procedure.

The following section describes the syntax for invoking a custom input procedure.

1.1 Syntax: /INPROCEDURE

To ensure your input procedure is called by **FieldShield**, use the following syntax within a job script:

```
/INPROCEDURE[=procedure_name]
[/PROCESS=process_type]
[/LENGTH=fixed_record_length] or [/LENGTH=0]
[other_attributes]
```

`/LENGTH=0` (the default) is used for variable-length records.

The default *`procedure_name`* is assumed to be `cs_input()`. Otherwise, the name specified by the calling program is used to resolve the procedure. The default process type is RECORD, which can be either fixed- or variable-length (records terminated by a linefeed or carriage return). The calling program can also specify an alternate process type, as described in /INREC *on page 161*.

The following sections describe the implementation process, in the order in which it is performed:

- Custom Input Procedure Declaration *on page 187*
- Building the Custom Input Procedure for Runtime Linking *on page 188*

1.2 Custom Input Procedure Declaration

A custom input procedures accepts two parameters:

- a pointer to the buffer that contains the input records

- a pointer to the size of the buffer in bytes.

The integer pointed to by the buffer size should be set to the actual number of bytes placed into the buffer by the input procedure.

The procedure(s) returns an integer value that should be set to 1, until there are no more records to pass. After the last record, the return value should be set to 0:

```
int procedure_name(char* buffer, int* buffersize)
```

The format of the data in the `buffer` depends on the `/PROCESS` specification in your **FieldShield** script (see /INREC *on page 161*).

1.3 Building the Custom Input Procedure for Runtime Linking

The custom procedures for **FieldShield** need to be built into a shared object. The defaults are **cs_user.so** (on UNIX/Linux) or a dynamic linked library (**cs_user.dll** on Windows). **FieldShield** looks for these default names in the current directory, and in the list of directories pointed to by the shared library path (UNIX/Linux) or the System32 folder (Windows).

When not using the default library names, the environment variable `CS_USER_DLL` must be used to point to the exact location of the library to be loaded.

For an example of a custom input procedure, see Example: Using Custom Input and Output Procedures *on page 190*.

NOTE

At runtime, **FieldShield** will attempt to resolve procedure names by looking for the defaults **cs_user.so** (UNIX/Linux) or **cs_user.dll** (Windows), and trying to resolve `CS_USER_DLL`.

2 Custom Output

In addition to the `/OUTFILE` option supported by **FieldShield** (see Output Files *on page 142*), you can also write and invoke your own custom output procedure. **FieldShield** allows the calling program to set up a custom output procedure by way of the command `/OUTPROCEDURE`.

You can use a custom output procedure to redirect sorted records to a proprietary DBMS or ETL tool, for example. In these cases, you can program a *hook* between the API of the tool (such as ODBC or OCI) and your **FieldShield** output procedure.

The following section describes the syntax for invoking a custom output procedure:

2.1 Syntax: /OUTPROCEDURE

To ensure your input procedure is called by **FieldShield**, use the following syntax:

```
/OUTPROCEDURE[=procedure_name]
[/PROCESS=process_type]
[/LENGTH=fixed_record_length] or [/LENGTH=0]
[other_attributes]
```

`/LENGTH=0` (the default) is used for variable-length records.

The default procedure name is assumed to be `cs_output()`. Otherwise, the name specified by the calling program is used to resolve the procedure. The default process type is RECORD, which can be either fixed- or variable-length (records terminated by a linefeed or carriage return). The calling program can also specify an alternate process type, as described in /INREC *on page 161*.

The following sections describe the implementation process, in the order in which it is performed:

- Custom Output Procedure Declaration *on page 189*.
- Building the Custom Output Procedure for Runtime Linking *on page 190*.

2.2 Custom Output Procedure Declaration

A custom output procedures accepts two parameters:

- a pointer to the buffer that contains one output record
- the length of that record.

After **FieldShield** has returned all the records, the value of the second parameter (the length) is set to NULL:

```
int procedure_name(char* buffer, int bufferlen)
```

The format of the data in the *`buffer`* depends on the `/PROCESS` specification in your **FieldShield** script (see /INREC *on page 161*).

2.3 Building the Custom Output Procedure for Runtime Linking

The custom procedures for **FieldShield** need to be built into a shared object. The defaults are **cs_user.so** (UNIX/Linux) or a dynamic linked library (**cs_user.dll**
on Windows). **FieldShield** looks for these defaults in the current directory, and in the list of directories pointed to by the shared library path (UNIX/Linux) or the System32 folder (Windows).

When not using the default library names, the environment variable `CS_USER_DLL` must be used to point to the exact location of the library to be loaded.

For an example of a custom input procedure, see Example: Using Custom Input and Output Procedures *on page 190*.

NOTE

At runtime, **FieldShield** will attempt to resolve procedure names by looking for the defaults **cs_user.so** (UNIX/Linux) or **cs_user.dll** (Windows), and trying to resolve `CS_USER_DLL`. Failure to resolve these procedures will result in an error.

3 Example: Using Custom Input and Output Procedures

This example illustrates the dynamic working of `/INPROCEDURE` and `/OUTPROCEDURE`.

Consider the following **FieldShield** job script, **customio.fcl**:

```
/INPROCEDURE=input_routine   # calling the inprocedure
/OUTPROCEDURE=output_routine # calling the outprocedure
```

This example also involves the input file **chiefs**, which contains information about United States Presidents (an excerpt follows):

```
...
Coolidge, Calvin          1923-1929   REP  VT
Hoover, Herbert C.        1929-1933   REP  IA
Roosevelt, Franklin D.    1933-1945   DEM  NY
Truman, Harry S.          1945-1953   DEM  MI
...
```

The following C program, **customio.c**, defines the input and output routines referenced in **customio.fcl** above:

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *
 *
 * inoutprocedures.c - example for dynamically testing /INPROC and /OUT-
PROC
 *
 * $Id: customio.c,v 1.5 2010/06/19 23:14:56 amrita Exp $
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *
 * athakur * Amrita Thakur * 2009-06-19
 *
 * Script:  customio.fcl
 * Input :  chiefs
 * Output:  stdout (via custom output routine)
 *
 * Copyright (c) 2009, Innovative Routines International, Inc.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * */

/*
 * define macros to allow exported functions under both Windows or UNIX
 */
#if defined (_WIN32)
#define _LIBSPEC __declspec (dllexport)
#include <windows.h>
#else
#define _LIBSPEC
#endif /* _WIN32 */

#include <stdio.h>
#include <string.h>
```

continued on next page...

```
FILE* ifile;

/* Custom Input Procedure called input_routine, which reads the
   input file "chiefs" and passes the data back to the calling
   program one record at a time.
*/
_LIBSPEC int input_routine(char* pBuffer, int* pLength) {
  int ret   = 0;
  static int first = 1;

  if (first){
    if ((ifile = fopen("chiefs","rb")) == (FILE*)0) {
      printf("error reading input file\n");
      ret = 0;
    }
    first = 0;
  }

  if (fgets(pBuffer, 500, ifile)) {
    *pLength = (int)strlen(pBuffer);
     ret = 1;
     printf(" in <%2d> |%.*s", *pLength, *pLength, pBuffer);
  }

  return (ret);
}

/* Custom Output Procedure called output_routine, which recieves a
   buffer from the calling program and sends it to "stdout".
*/
_LIBSPEC int output_routine(char* pBuffer, int iLength) {
  char* ptr;
  int   iLen;

  if (pBuffer){
    pBuffer[iLength] = '\0';
    printf("Output buffer size is  %d\n\n%s\n", iLength, pBuffer);
  }

  return 0;
```

FieldShield Metadata Tools

This chapter discusses the tools that aid in the use of the **FieldShield** program.
It contains the following sub-chapters:

- cob2ddf *on page 194*
- csv2ddf *on page 196*
- ctl2ddf *on page 198*
- elf2ddf *on page 200*
- ldif2ddf *on page 203*
- xml2ddf *on page 205*
- ldif2ddf *on page 203*

These command line programs parse COBOL copybooks, comma-separated values files, SQL*Loader control file, extended web logs, LDIF layouts, XML layouts, and database tables (respectively) to generate **FieldShield** data definition files that can be referenced repeatedly in different **FieldShield** job scripts.

NOTE

Meta Integration Technology, Inc. (MITI) has also created a Meta Integration Model Bridge (MIMB) that automatically converts flat file layouts formatted in third-party application metadata structures (e.g., DataStage .dsx, Informatica .xml, and XMI) into **FieldShield** data definition files (.ddf). For more information on available metadata conversions, see:
`http://metaintegration.net/Products/MIMB`.

- CLF Templates *on page 211*

  Three **.ddf** files are provided as metadata support for the three NCSA separate (common) log file types. This facilitates the use of **FieldShield** for protecting sensitive field data within eCommerce web logs saved in common log format.

1 cob2ddf

cob2ddf (COBOL-to-DDF) is a translation program for users with input data from a COBOL application who want to convert the record (copybook) layouts into **FieldShield** data definition files. The **cob2ddf** program, located in the ***install_dir*\bin** directory on Windows (**$FIELDSHIELD_HOME/bin** on UNIX and Linux), produces descriptive file name and field-layout text that can be referenced by, or pasted directly into, a **FieldShield** job script.

Currently, **cob2ddf** does not convert the entire range of COBOL data-definition functionality, but it does provide a convenient way to convert field descriptions for use in **FieldShield**. See the Micro Focus COBOL Language Reference for documentation on the data description portion of COBOL programs.

1.1 Usage

The syntax is:

```
cob2ddf [-warningsoff] [-use-wordstorage] infile [ddfile]
```

To execute **cob2ddf** and create a **FieldShield** data definition file, enter:

```
cob2ddf filename.cbl ddf_file
```

where `filename.cbl` is the name of the copybook or file description file, and `ddf_file` is the resultant **FieldShield** data definition file. For example, the command:

```
cob2ddf cpybk.cbl cpybk.ddf
```

converts the file and field descriptions in **cpybk.cbl** to a **FieldShield** data definition file named **cpybk.ddf**. For details on how to reference a **.ddf** file from within a **FieldShield** job script, see Data Definition Files *on page 143*.

You can turn off any runtime warning messages using the `-warningsoff` flag, for example:

```
cob2ddf -warningsoff cpybk.cbl cpybk.ddf
```

By default, runtime warning messages are enabled.

You can add a `wordstorage` flag on the command line to ensure that **FieldShield** field `SIZE` attributes are computed in the word storage mode, for example:

```
cob2ddf  –use-wordstorage copybk.cbl cpybk.ddf
```

1.2 Example

The following is a COBOL copybook file, **cob.app**:

```
        01 REG
           05 PLANT PIC X(08).
           05 FREE PIC 9(10).
           05 CLIENT PIC X(09).
           05 CARRIER-18 PIC 9(12).
           05 CARRIER-23PIC 9(12).
           05 INTEREST PIC S9(11) sign is leading
                             separate character.
           05 CARGOES PIC S9(12) sign is leading
                             separate character.
```

To create the **FieldShield** equivalent, enter the following on the command line:

```
cob2ddf cob.app cob.ddf
```

The resultant data definition file, **cob.ddf**, is:

```
    /FIELD=(FREE, POSITION=9, SIZE=10, NUMERIC)
    /FIELD=(CLIENT, POSITION=19, SIZE=9)
    /FIELD=(CARRIER_18, POSITION=28, SIZE=12, NUMERIC)
    /FIELD=(CARRIER_23, POSITION=40, SIZE=12, NUMERIC)
    /FIELD=(INTEREST, POSITION=52, SIZE=12, MF_DISPSLS)
    /FIELD=(CARGOES, POSITION=64, SIZE=13, MF_DISPSLS)
```

This file gives the record layout (field definitions) for a file that can be used for input
or for output. See Data Definition Files *on page 143* for details on using a **.ddf** within a **FieldShield** job script.

2 csv2ddf

csv2ddf (comma separated values-to-DDF) is a translation program for converting Microsoft CSV file header descriptions to **FieldShield** data definition files. The **csv2ddf** program, located in the ***install_dir*\bin** directory on Windows (**$FIELDSHIELD_HOME/bin** on UNIX and Linux), scans CSV files to produce descriptive file name and input field layout text from the header that can be referenced by, or pasted directly into, a **FieldShield** job specification file.

NOTE For the purposes of **csv2ddf**, it is expected that the first record of a Microsoft **.csv** file data is preceded by header descriptions. Therefore, if your data file does not have a header, you cannot use the **csv2ddf** program. See SEPARATOR *on page 150* for details on how to use **FieldShield** to define input file fields for comma-delimited records.

For details on using the `/PROCESS=CSV` command in a **FieldShield** specification file to instruct **FieldShield** to treat a file as a **.csv** file, see CSV *on page 172*.

2.1 Usage

The syntax of **csv2ddf** is:

```
csv2ddf filename.csv filename.ddf
```

where `filename.csv` is the name of the comma-separated-values file, and `filename.ddf` is the resulting **FieldShield** data definition file. For example, the command:

```
csv2ddf spreadsheet.csv spreadsheet.ddf
```

converts the field layout descriptions in `spreadsheet.csv` to a **FieldShield** data definition file called **spreadsheet.ddf**. For details on how to reference a **.ddf** file from within a **FieldShield** job script, see Data Definition Files *on page 143*.

2.2 Example

Using the following CSV input file, **test.csv**:

```
Element_Name,Windows_NT,Windows,Windows_CE,Win32s,Component,
Component_Version,Header_File,Import_Library,Unicode,Element_Type
ADsBuildEnumerator,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsBuildVarArrayInt,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsBuildVarArrayStr,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsEnumerateNext,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsFreeEnumerator,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsGetLastError,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsGetObject,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsOpenObject,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsSetLastError,4.0 or later,,,,ADSI,,adshlp.h,,,function
IADs::Get,4.0 or later,,,,ADSI,,iads.h,,,interface method
```

Execute the following command to generate the **.ddf**:

```
csv2ddf test.csv csv.ddf
```

You can then modify the resultant **csv.ddf** to create the following script, **csv.fcl**:

```
/INFILE=e:\test.csv
/PROCESS=CSV
   /LENGTH=0
   /FIELD=(Element_Name,POSITION=1,SEPARATOR=',')
   /FIELD=(Windows_NT,POSITION=2,SEPARATOR=',')
   /FIELD=(Windows,POSITION=3,SEPARATOR=',')
   /FIELD=(Windows_CE,POSITION=4,SEPARATOR=',')
   /FIELD=(Win32s,POSITION=5,SEPARATOR=',')
   /FIELD=(Component,POSITION=6,SEPARATOR=',')
   /FIELD=(Component_Version,POSITION=7,SEPARATOR=',')
   /FIELD=(Header_File,POSITION=8,SEPARATOR=',')
   /FIELD=(Import_Library,POSITION=9,SEPARATOR=',')
   /FIELD=(Unicode,POSITION=10,SEPARATOR=',')
   /FIELD=(Element_Type,POSITION=11,SEPARATOR=',')
/OUTFILE=e:\csv.out
   /LENGTH=0
```

Because the file type is converted from `/PROCESS=CSV` to `/PROCESS=RECORD`, the header record is removed from the sorted output.

You can also specify `/PROCESS=CSV` on output, in which case **FieldShield** adds a CSV-style header record based on the field names specified on output (or the field names specified on input if there is no remapping).

When specifying `/PROCESS=CSV` on output, **FieldShield** will enclose each output field with double-quotes if you specify comma-separated fields on output, which is recommended. Otherwise, the input field layout is used by default (which may not be comma-separated), and the only change on output will be the addition of the header.

3 ctl2ddf

The program **ctl2ddf** allows **FieldShield** users to convert the column layouts specified in a SQL*Loader control file (**.ctl**) into a data definition file (**.ddf**) containing `/FIELD` layout descriptions. This **.ddf** can then be used in **FieldShield** job scripts (see Data Definition Files *on page 143*). The **ctl2ddf** program is located in the **\install_dir\bin** directory on Windows (**$FIELDSHIELD_HOME/bin** on UNIX and Linux).

3.1 Usage

You must have permission to access both the **ctl2ddf** and **FieldShield** programs. These are standalone programs, i.e., they are not interdependent and do not require any other modules.

To execute **ctl2ddf**, enter:

```
ctl2ddf control_file
```

where *`control_file`* is the SQL*Loader control file (typically **.ctl**) containing the column layout specifications you want to convert. The output is sent to ***filename*.ddf**, where ***`filename`*** is the table name specified in the **.ctl** file into which the data will be loaded.

3.2 Example

Given the following SQL*Loader control file, **test1.ctl**:

```
# SQL*Loader Control File specifications (user)
# Fri Mar 30 17:LOAD DATA
INFILE 'out.dat'
INTO TABLE emp2
TRAILING NULLCOLS
(EMPNO position(0001:0006) DECIMAL EXTERNAL NULLIF (EMPNO = BLANKS),
ENAME position(0007:0016) char,
JOB  position(0017:0026) char,
MGR position(0027:0032) DECIMAL EXTERNAL NULLIF (MGR = BLANKS),
SAL position(0033:0043) DECIMAL EXTERNAL NULLIF (SAL = BLANKS),
COMM  position(0044:0053) DECIMAL EXTERNAL NULLIF (COMM = BLANKS),
DEPTNO position(0054:0056) DECIMAL EXTERNAL NULLIF(DEPTNO = BLANKS))
```

The command to convert it to a **FieldShield** data definition file might be:

```
ctl2ddf test1.ctl
```

emp2.ddf is generated:

```
/FILE=out.dat
/FIELD=(EMPNO, POSITION=1, SIZE=6, DOUBLE)
/FIELD=(ENAME, POSITION=7, SIZE=10)
/FIELD=(JOB, POSITION=17, SIZE=10)
/FIELD=(MGR, POSITION=27, SIZE=6, DOUBLE)
/FIELD=(SAL, POSITION=33, SIZE=11, DOUBLE)
/FIELD=(COMM, POSITION=44, SIZE=10, DOUBLE)
/FIELD=(DEPTNO, POSITION=54, SIZE=3, DOUBLE)
```

Note that the `INFILE` source for the **.ctl** file, **out.dat**, is used as the `/FILE` specification in the **.ddf** (see in Data Definition Files *on page 143*).

This **.ddf** can be now invoked from within a **FieldShield** job script, for example:

```
/SPECIFICATION=test1.ddf
/INFILE=out.dat
/OUTFILE=out_reduced.dat
    /FIELD=(EMPNO, POSITION=1, SIZE=6, DOUBLE)
    /FIELD=(MGR, POSITION=7, SIZE=6,DOUBLE)
```

4 elf2ddf

elf2ddf (extended log format-to-DDF) is a translation program for converting W3C web data descriptions to **FieldShield** data definition files. The **elf2ddf** program, located in the **\*install_dir*\bin** directory on Windows (**$FIELDSHIELD_HOME/bin** on UNIX and Linux), scans web transaction files in ELF to produce descriptive file name and field-layout text from the header that can be referenced by, or pasted directly into, a **FieldShield** job specification file.

For details on using the `/PROCESS=ELF` command in the `/INFILE` and `/OUTFILE` sections of a **FieldShield** specification file, see ELF (W3C Extended Log Format) *on page 181*.

FieldShield also handles web logs in CLF format. The sample **FieldShield** data definition files **CLF_Referrer.ddf**, **CLF_Agent.ddf**, and **CLF_Access.ddf** are provided in the **examples/fieldshield** directory.

4.1 Usage

elf2ddf requires that your ELF file is in the W3C convention format described on this page.

An extended log file contains a sequence of lines containing ASCII characters terminated by either the sequence LF or CRLF. Log file generators should follow the line termination convention for the platform on which they are executed. Analyzers should accept either form. Each line may contain either a *directive* or an *entry*.

Entries consist of a sequence of fields relating to a single HTTP transaction. Fields are separated by white space, the use of tab characters for this purpose is encouraged. If a field is unused in a particular entry dash "-" marks the omitted field. Directives record information about the logging process itself.

Lines beginning with the # character contain directives. The following directives are defined:

- `Version:` *integer.integer*
  The version of the extended log file format used.
- `Fields:` [*specifier...*]
  Specifies the fields recorded in the log.
- `Software:` *string*
  Identifies the software which generated the log.
- `Start-Date:` *date_time*
  The date and time at which the log was started.

- `End-Date:` *date_time*
  The date and time at which the log was finished.
- `Date:` *date_time*
  The date and time at which the entry was added.
- `Remark:` *text*
  Specifies comment information. Data recorded in this field should be ignored by analysis tools.

The directives `Version` and `Fields` are required and should precede all entries in the log. The `Fields` directive specifies the data recorded in the fields of each entry.

The syntax of **elf2ddf** is:

```
elf2ddf filename.elf filename.ddf
```

To execute **elf2cl** and create a **FieldShield** data definition file, enter:

```
elf2ddf filename.elf fieldshield_ddf
```

where *`filename`*`.elf` is the name of the extended log format file, and *`fieldshield_ddf`* is the resulting **FieldShield** data definition file. For example, the command:

```
elf2ddf clickstream.elf clickstream.ddf
```

converts the field layout descriptions in the **clickstream.elf** file header to a **FieldShield** data definition file named **clickstream.ddf**. For details on how to reference a **.ddf** file from within a **FieldShield** job script, see Data Definition Files *on page 143*.

4.2 Example

The following is a header from an ELF file:

```
#Version 1.0
#Date:  12-Jan-2000 00:00:00
#Fields: time cs-method cs-url
00:24:23 GET /tak/far.html
12:21:16 GET /tak/far.html
12:45:52 GET /tak/far.html
12:57:34 GET /tak/far.html
```

The following is the **FieldShield .ddf** generated by **elf2ddf** based on the above header:

```
# FIELDSHIELD DDF for ELF data
# Generated by elf2ddf.exe based on ELF header
# in "data.elf".
/file=data.elf
/process=ELF
   /length=0
   /field=(time, position=1, separator=' ', ASCII)
   /field=(cs-method, position=2, separator=' ', ASCII)
   /field=(cs-url, position=3, separator=' ', ASCII)
```

NOTE

If you specify `/PROCESS=ELF` on output in a **FieldShield** job script, an ELF-style header will be generated.

5 ldif2ddf

The program **ldif2ddf** allows **FieldShield** users to convert the column layouts specified in an LDIF (Lightweight Directory Interchange) format into a data definition file (**.ddf**) containing `/FIELD` layout descriptions. This **.ddf** can then be used in **FieldShield** job scripts (see Data Definition Files *on page 143*). LDIF is the format of data exported from an LDAP database.

The **ldif2ddf** program is located in the directory **$FIELDSHIELD_HOME/bin** on UNIX (\install_dir\bin on Windows).

5.1 Usage

The syntax of **ldif2ddf** is:

```
ldif2ddf [-s separator] source_filename [target_filename]
```

where *source_filename* is the name of the LDIF file, and *target_filename* is the resultant **FieldShield** data definition file (typically with a **.ddf** extension). If no target file is specified, the data definition statements will be written to the console.

Optionally, you can specify:

-s *separator*	where *separator* is a field SEPARATOR string that you specify (see SEPARATOR *on page 150*). The default separator string is a pipe (|). The SEPARATOR and POSITION attributes are required for LDIF input fields defined in **FieldShield**, but they have only internal significance (as described in LDIF *on page 172*).

ldif2ddf translates all of the LDIF fields that are found in the source LDIF file, so you may want to manually delete unwanted elements in the target DDF.

5.2 Example

The following is an excerpt from the file **plant_catalog.ldif**:

```
common: Phlox, Woodland
botanical: Phlox divaricata
zone: 3
light: Sun or Shade
price: $2.80
availability: Jan/22/2008

common: Cardinal Flower
botanical: Lobelia cardinalis
zone: 2
light: Shade
price: $3.02
availability: Feb/22/2008

common: California Poppy
botanical: Eschscholzia californica
zone: Annual
light: Sun
price: $7.89
availability: Mar/27/2008

common: Mayapple
botanical: Podophyllum peltatum
zone: 3
light: Mostly Shady
price: $2.98
availability: Jun/05/2008
```

The following is the **FieldShield** DDF translation produced by running **ldif2ddf**:

```
/FIELD=(common, POSITION=1, SEPARATOR='|')
/FIELD=(botanical, POSITION=2, SEPARATOR='|')
/FIELD=(zone, POSITION=3, SEPARATOR='|')
/FIELD=(light, POSITION=4, SEPARATOR='|')
/FIELD=(price, POSITION=5, SEPARATOR='|')
/FIELD=(availability, POSITION=6, SEPARATOR='|')
```

This DDF can now be referenced in a **FieldShield** job script.

6 xml2ddf

xml2ddf (eXtensible Markup Language format-to-DDF) is a translation program for scanning XML files to produce descriptive file name and field-layout text that can be referenced by, or pasted directly into, a **FieldShield** job specification file The **xml2ddf** program is located in the **\*install_dir*\bin** directory on Windows (**$FIELDSHIELD_HOME/bin** on UNIX and Linux).

After translation, you can edit the format of the field statements within the **FieldShield** data definition file, or within the **FieldShield** job specification file if the DDF contents have been pasted or appended.

For details on how to reference a **.ddf** file from within a **FieldShield** job script, see Data Definition Files *on page 143*.

For details on using the `/PROCESS=XML` command in the `/INFILE` and `/OUTFILE` sections of a **FieldShield** specification file, see XML *on page 180*.

6.1 Usage

The syntax of **xml2ddf** is:

```
xml2ddf [options] source_filename [target_filename]
```

where *`source_filename`* is the name of the XML file (typically with a **.xml** extension), and *`target_filename`* is the resultant **FieldShield** data definition file (typically with a **.ddf** extension). If no target file is specified, the data definition statements will be written to the console.

Additional options include:

-f *frame*	where *frame* is a `FRAME` character that you specify (see FRAME *on page 155*). The default frame character is a double quote (").
-s *separator*	where *separator* is a field `SEPARATOR` string that you specify (see SEPARATOR *on page 150*). The default separator string is a pipe (|). The `SEPARATOR` and `POSITION` attributes are required for XML input fields defined in **FieldShield**, but they have only internal significance (as described in XML *on page 180*).

xml2ddf translates all of the XML fields that are found in the source XML file, so you may want to manually delete unwanted elements in the target DDF.

6.2 Example

The following is an excerpt from the file **plant_catalog.xml**:

```
<CATALOG>
        <PLANT>
                <COMMON>Bloodroot</COMMON>
                <BOTANICAL>Sanguinaria canadensis</BOTANICAL>
                <ZONE>4</ZONE>
                <LIGHT>Mostly Shady</LIGHT>
                <PRICE>$2.44</PRICE>
                <AVAILABILITY>2008-03-15</AVAILABILITY>
        </PLANT>
...
        <PLANT>
                <COMMON>Don&apos;s Flower</COMMON>
                <BOTANICAL>Thornicus purnhagenus</BOTANICAL>
                <ZONE>Perennial</ZONE>
                <LIGHT>Sun &amp; Shade</LIGHT>
                <PRICE>$99.95</PRICE>
                <AVAILABILITY>2008-08-15</AVAILABILITY>
        </PLANT>
</CATALOG>
```

The following is the **FieldShield** DDF translation produced by running **xml2ddf.exe**:

```
/FIELD=(COMMON0001, POS=0001, SEP='|', FRAME='"', XDEF="/CATALOG/PLANT/COMMON")
/FIELD=(BOTANICAL0002, POS=0002, SEP='|', FRAME='"', XDEF="/CATALOG/PLANT/BOTANICAL")
/FIELD=(ZONE0003, POS=0003, SEP='|', FRAME='"', XDEF="/CATALOG/PLANT/ZONE")
/FIELD=(LIGHT0004, POS=0004, SEP='|', FRAME='"', XDEF="/CATALOG/PLANT/LIGHT")
/FIELD=(PRICE0005, POS=0005, SEP='|', FRAME='"', XDEF="/CATALOG/PLANT/PRICE")
/FIELD=(AVAILABILITY0006, POS=0006, SEP='|', FRAME='"', XDEF="/CATALOG/PLANT/AVAILABILITY")
```

7 odbc2ddf

The program **odbc2ddf** allows **FieldShield** users to convert database table layouts into a data definition file (**.ddf**) containing `/FIELD` layout descriptions, provided the database is compatible with ODBC (Open Database Connectivity). This **.ddf** can then be used in **fieldshield** job scripts (see Data Definition Files *on page 143*). ODBC is an interface that accesses certain databases.

odbc2ddf also allows you to query the database for such information as the list of ODBC drivers,

The **odbc2ddf** program is located in the directory **$FIELDSHIELD_HOME/bin** on UNIX (**\*install_dir*\bin** on Windows).

7.1 Basic Syntax

The syntax for basic usage of **odbc2ddf**, that is, to generate `/FIELD` statements from a table description is:

```
dbc2ddf [Table] [outfile]
```

where `Table` is the name of an RDBMS table, and *`outfile`* is the resultant **fieldshield** data definition file (typically with a **.ddf** extension). If no target file is specified, data definition statements are written to the console.

7.2 Complete Syntax

Complete syntax for the **odbc2ddf** command is:

```
odbc2ddf [-d] [-u username] [-p password] [DSN] [Table] [outfile]
          [-f frame] [-s separator] [-l recordlength]
```

where:

-d	Lists the installed ODBC drivers
-u	User name for the specified DSN. If no DSN is given, the possible DSNs are listed. If no `Table` is given, the tables in the specified DSNs are listed.
-p	Password for the specified DSN.
DSN	The name of the DSN.
Table	A specific table name from which to generate /FIELD statements.

outfile	The resultant **fieldshield** data definition file (typically with a **.ddf** extension). If no `outfile` is specified, the data definition statements will be written to the console.
-f	Specifies the frame character used in the `/FIELD` statement.
-s	Specifies the separator string used in the `/FIELD` statement
-l	Output fixed length fields.

Additional options include:

-h	Displays help.
-v	Verbose mode.

odbc2ddf translates all of the table columns that are found in the source table, so you may want to manually delete unwanted columns (/FIELDs) in the target file.

7.3 Example

The following is a database description of a table, **tablea**, residing on the database with DSN **my_oracle**:

Field	Type	Null	Default
name	varchar(40)	No	
age	smallint(5)	No	0
address	mediumint(6)	No	0
st	varchar(2)	No	
zip	text	No	
rate	float(4,2)	No	0.00
year	year(4)	No	0000
date	date	No	0000-00-00
timestamp	timestamp	Yes	CURRENT_TIMESTAMP
time	time	No	00:00:00
active	tinyint(1)	No	0
offset	int(11)	No	0
key	var(64)	No	

Field	Type	Null	Default
spanish	varchar(8)	No	
adjust	double(8,3)	No	0.00
filled	int(8)	No	00000099

The following is the **fieldshield** DDF translation produced by running the following:

odbc2ddf my_oracle tablea table.ddf:

```
/FILE="tablea;DSN=my_oracle"
    /PROCESS=ODBC
    /FIELD=(name, POSITION=0001, SEPARATOR='|')
    /FIELD=(age, POSITION=0002, SEPARATOR='|', NUMERIC)
    /FIELD=(address, POSITION=0003, SEPARATOR='|', NUMERIC)
    /FIELD=(st, POSITION=0004, SEPARATOR='|')
    /FIELD=(zip, POSITION=0005, SEPARATOR='|')
    /FIELD=(rate, POSITION=0006, SEPARATOR='|', NUMERIC)
    /FIELD=(year, POSITION=0007, SEPARATOR='|', NUMERIC)
    /FIELD=(date, POSITION=0008, SEPARATOR='|')
    /FIELD=(timestamp, POSITION=0009, SEPARATOR='|')
    /FIELD=(time, POSITION=0010, SEPARATOR='|')
    /FIELD=(active, POSITION=0011, SEPARATOR='|')
    /FIELD=(offset, POSITION=0012, SEPARATOR='|', NUMERIC)
    /FIELD=(key, POSITION=0013, SEPARATOR='|')
    /FIELD=(spanish, POSITION=0014, SEPARATOR='|')
    /FIELD=(adjust, POSITION=0015, SEPARATOR='|', NUMERIC)
    /FIELD=(filled, POSITION=0016, SEPARATOR='|', NUMERIC)
```

Note that the comments in the above example refer to the ODBC data type value, which can be any of the following that have **fieldshield** data type equivalents:

Table 19: Supported ODBC Data Types

ODBC Data Type	FieldShield Equivalent
CHAR	ASCII
NUMERIC	NUMERIC
DECIMAL	NUMERIC
INTEGER	NUMERIC, PRECISION=0
SMALLINT	NUMERIC, PRECISION=0
FLOAT	NUMERIC

Table 19: Supported ODBC Data Types

ODBC Data Type	FieldShield Equivalent
REAL	NUMERIC
DOUBLE	NUMERIC
DATETIME	ASCII
VARCHAR	ASCII
DATE	ISO_DATE or ASCII
TIME	ISO_TIME or ASCII
TIMESTAMP	ISO_TIMESTAMP or ASCII

8 CLF Templates

NCSA Separate log (or *three-log*) format is a convention for storing NCSA common log data in three separate log files, rather than as a single file. The three log formats are:

Common (Access) log	Contains basic information from the NCSA log.
Referral log	Contains corresponding referral information.
Agent log	Contains corresponding agent information.

FieldShield furnishes metadata support for these formats by providing three data definition file (**.ddf**) templates, each containing `/FIELD` layout descriptions usable in **FieldShield** job scripts (see Data Definition Files *on page 143*).

8.1 Usage

In the **\*install_dir*\examples** directory on Windows (**$FIELDSHIELD_HOME/ examples** on UNIX and Linux), three **.ddf** templates with NCSA log field layouts are provided:

CLF_Access.ddf	**FieldShield** field layouts for common log or access log files.
CLF_Referral.ddf	**FieldShield** field layouts for referral log files.
CLF_Agent.ddf	**FieldShield** field layouts for agent log files.

To create job scripts which reference these layouts, you can open the **.ddf** you require, modify it to include job statements and any output file specifications, and save it as an **.fcl** job script that can run with **FieldShield** (see Examples *on page 211*).

Alternatively, you can create a new job script, and use a /SPEC statement to invoke the field layouts from the **.ddf** you require (see Specification Files *on page 144*).

8.2 Examples

8.2.1 Common (Access) Log

Given the NCSA common log file input data, **common.dat**:

```
129.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
127.125.125.125 - rjones [10/Oct/1999:21:15:04 +0500] "GET /index.html HTTP/1.0" 200 1043
127.125.125.125 - dsmith [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
125.125.125.125 - dsmith [10/Oct/1999:21:15:02 +0500] "GET /index.html HTTP/1.0" 200 1043
128.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
129.125.125.125 - dsmith [10/Oct/1999:21:15:08 +0500] "GET /index.html HTTP/1.0" 200 1043
125.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
126.125.125.125 - rjones [10/Oct/1999:21:15:04 +0500] "GET /index.html HTTP/1.0" 200 1043
```

```
126.125.125.125 - dsmith [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
128.125.125.125 - dsmith [10/Oct/1999:21:15:02 +0500] "GET /index.html HTTP/1.0" 200 1043
```

You can open **CLF_Access.ddf**, and modify it to create **common.fcl**:

```
/INFILE=common.dat
    /FIELD=(host,POSITION=1,SEPARATOR=' ',IP_ADDRESS)
    /FIELD=(rfc931,POSITION=2,SEPARATOR=' ')
    /FIELD=(username,POSITION=3,SEPARATOR=' ')
    /FIELD=(time,POSITION=4,SEPARATOR=' ',EUROPEAN_TIMESTAMP)
    /FIELD=(timezone,POSITION=5,SIZE=5,SEPARATOR=' ')
    /FIELD=(request,POSITION=6,SEPARATOR=' ',FRAME='"')
    /FIELD=(statuscode,POSITION=7,SEPARATOR=' ',NUMERIC)
    /FIELD=(bytes,POSITION=8,SEPARATOR=' ',NUMERIC)
/OUTFILE=common.csv
    /PROCESS=CSV
    /FIELD=(host,POSITION=1,SEPARATOR=’,’,IP_ADDRESS)
    /FIELD=(rfc931,POSITION=2,SEPARATOR=’,’)
    /FIELD=(username,POSITION=3,SEPARATOR=' ')
    /FIELD=(time,POSITION=4,SEPARATOR=’,’,EUROPEAN_TIMESTAMP)
    /FIELD=(timezone,POSITION=5,SIZE=5,SEPARATOR=’,’)
```

When executed, this produces the following reduced output in CSV format:

```
host,rfc931,username,time,timezone
"129.125.125.125","-" "rjones","[10/Oct/1999:21:15:07","+0500"
"127.125.125.125","-" "rjones","[10/Oct/1999:21:15:04","+0500"
"127.125.125.125","-" "dsmith","[10/Oct/1999:21:15:07","+0500"
"125.125.125.125","-" "dsmith","[10/Oct/1999:21:15:02","+0500"
"128.125.125.125","-" "rjones","[10/Oct/1999:21:15:07","+0500"
"129.125.125.125","-" "dsmith","[10/Oct/1999:21:15:08","+0500"
"125.125.125.125","-" "rjones","[10/Oct/1999:21:15:07","+0500"
"126.125.125.125","-" "rjones","[10/Oct/1999:21:15:04","+0500"
"126.125.125.125","-" "dsmith","[10/Oct/1999:21:15:07","+0500"
"128.125.125.125","-" "dsmith","[10/Oct/1999:21:15:02","+0500"
```

8.2.2 Referral Log

Given the NCSA referral log file input data, **ref.dat**:

```
[04/Sep/2004:21:15:05 +0500] "http://www.ibm.com"
[12/Oct/2004:21:18:05 +0500] "http://www.aol.com/index.html"
[22/Nov/2004:21:15:05 +0500] "http://www.microsoft.com"
[29/Dec/2004:21:16:05 +0500] "http://www.ibm.com/index.html"
[19/Oct/2004:21:15:05 +0500] "http://www.apple.com"
[04/Nov/2004:21:15:05 +0500] "http://www.aol.com/index.html"
[10/Oct/2004:21:15:05 +0500] "http://www.microsoft.com/index.html"
[10/Oct/2004:21:15:01 +0500] "http://www.ibm.com/index.html"
[10/Oct/2004:21:15:05 +0500] "http://www.apple.com/index.html"
[10/Oct/2004:21:15:05 +0500] "http://www.aol.com"
```

You can open **CLF_Referral.ddf**, and modify it to create **referral.fcl**:

```
/INFILE=ref.dat
    /FIELD=(Time,POSITION=2,SIZE=20,EUROPEAN_TIMESTAMP) # fixed field
    /FIELD=(Referrer,POSITION=3,SEPARATOR=' ',FRAME='"') # delimited field
/OUTFILE=referral.out
    /FIELD=(Referrer,POSITION=1,SEPARATOR=',')
    /FIELD=(Time,POSITION=2,SEPARATOR=',',EUROPEAN_TIMESTAMP,SIZE=12)
/DATA={9}"*"
```

When executed, this produces the following protected output, **referral.out**:

```
http://www.ibm.com,04/Sep/2004:*********
http://www.aol.com/index.html,12/Oct/2004:*********
http://www.microsoft.com,22/Nov/2004:*********
http://www.ibm.com/index.html,29/Dec/2004:*********
http://www.apple.com,19/Oct/2004:*********
http://www.aol.com/index.html,04/Nov/2004:*********
http://www.microsoft.com/index.html,10/Oct/2004:*********
http://www.ibm.com/index.html,10/Oct/2004:*********
http://www.apple.com/index.html,10/Oct/2004:*********
http://www.aol.com,10/Oct/2004:*********
```

8.2.3 Agent Log

Given the NCSA agent log file input data, **agent.dat**:

```
[10/Nov/2004:21:10:05 +0500] "Microsoft Internet Explorer - 5.0"
[15/Oct/2004:20:11:05 +0500] "Microsoft Internet Explorer - 6.0"
[16/Oct/2004:22:15:07 +0500] "Microsoft Internet Explorer - 6.0"
[09/Nov/2004:09:15:25 +0500] "Microsoft Internet Explorer - 5.0"
[11/Nov/2004:21:17:35 +0500] "Microsoft Internet Explorer - 5.0"
[11/Oct/2004:01:35:25 +0500] "Microsoft Internet Explorer - 6.0"
[10/Oct/2004:11:11:15 +0500] "Microsoft Internet Explorer - 6.0"
[05/Nov/2004:23:15:45 +0500] "Microsoft Internet Explorer - 5.0"
[14/Oct/2004:03:15:05 +0500] "Microsoft Internet Explorer - 6.0"
[03/Nov/2004:19:25:45 +0500] "Microsoft Internet Explorer - 5.0"
```

You can open **CLF_Agent.ddf**, and modify it to create **agent.fcl**:

```
/INFILE=agent.dat
    /FIELD=(time,POSITION=2,SIZE=20,EUROPEAN_TIMESTAMP) # fixed field
    /FIELD=(agent,POSITION=3,SEPARATOR=' ',FRAME='"')   # delimited field
/OUTFILE=agent.csv
    /FIELD=(time,POSITION=1,SEPARATOR=',',EUROPEAN_TIMESTAMP,SIZE=12)
 /DATA={9}"*"
/FIELD=(agent,POSITION=2,SEPARATOR=',')
```

When executed, this produces the following output:

```
10/Nov/2004:*********,Microsoft Internet Explorer - 5.0
15/Oct/2004:*********,Microsoft Internet Explorer - 6.0
16/Oct/2004:*********,Microsoft Internet Explorer - 6.0
09/Nov/2004:*********,Microsoft Internet Explorer - 5.0
11/Nov/2004:*********,Microsoft Internet Explorer - 5.0
11/Oct/2004:*********,Microsoft Internet Explorer - 6.0
10/Oct/2004:*********,Microsoft Internet Explorer - 6.0
05/Nov/2004:*********,Microsoft Internet Explorer - 5.0
14/Oct/2004:*********,Microsoft Internet Explorer - 6.0
03/Nov/2004:*********,Microsoft Internet Explorer - 5.0
```

Upgrade Options

FieldShield is part of the IRI Data Manager Suite of products. Additional IRI tools are available to run in conjunction with -- or as an immediately available and metadata-compatible upgrade from – **FieldShield**. All IRI software products share the same metadata and Eclipse GUI (the IRI Workbench).

CoSort (Big Data Transformation & Reporting)

- **CoSort** is a high-performance data manipulation package for "big data" in files and tables, which can simultaneously transform, convert, protect, and report. **CoSort** is often used in high-volume DW/BI environments, legacy migrations, SQL/shell/3GL program replacements, and third-party sort upgrades.
- **CoSort** allows DW/BI architects to simultaneously filter, sort, join, aggregate, cross-calculate, sequence, protect, and remap data in most DB and flat file formats (or new data sources through custom procedures), cutting the time and cost of data integration and migration.
- **CoSort** can map fields from sources to targets for ETL and DB migration, generate table creation and loader metadata, and speed bulk loads by pre-sorting flat files on the longest index key of the table.
- **CoSort** can also: segment data and capture changes; write detail and summary data in custom reports; handle slowly changing dimensions; pivot and unpivot; map to bespoke schemas; spot trends via lookups and functions; use 3rd-party data quality engines; and, protect private data at the field level.
- **CoSort** provides drop-in sort replacement or sort parm conversion routines for DataStage, DB2 Load, Informatica, Micro Focus COBOL, MVS & VSE JCL, SAS, Software AG Natural, and UNIX.

FACT (Fast Extract)

FACT is high-performance unload utility for Oracle, DB2, Sybase, SQL Server, MySQL, Altibase, and Tibero. **FACT** is typically used in data warehouse extract-transform-load (ETL) and ELT operations, database archiving and migration scenarios, and to improve the performance of offline reorg operations.

- **FACT** extracts huge tables to flat files in parallel, using simple config files with SQL SELECT queries.
- **FACT** features built-in data conversion and reformatting options to change data types and layouts.
- **FACT** writes the extract metadata for CoSort staging and reporting scripts, and for bulk, pre-CoSorted, loads, making FACT an essential acquisition component in big data integration and staging.
- **FACT** runs with CoSort and DB loaders in the IRI Workbench (Eclipse GUI) for offline reorg and ETL.

RowGen (Test Data)

RowGen builds structurally and referentially correct test data in the same form and format of real tables, files, and reports – without real data. **RowGen** is used to build and test applications, populate data warehouses, simulate and outsource file and report formats, and build benchmarking data sets when production data is classified or unavailable.

- **RowGen** simulates production tables, files, and reports with realistic and referentially correct test data faster than any other method.
- **RowGen** automatically parses data models, generates the test data, and populates its targets all at once.
- **RowGen** combines random data generation, lookup table selection, and data manipulation to enhance test data realism and protect privacy.
- **RowGen** uses your data models, the metadata in CoSort (and all IRI tools), or Meta Integration Model Bridge (MIMB) repositories, to match your production formats.

NextForm

NextForm converts, replicates, and federates data in databases and files. **NextForm** is typically used to migration data from mainframes and legacy databases for use in new applications and platforms.

- **NextForm** moves and maps data between DB2, MySQL, Oracle, SQL Server and Sybase, and facilitates new database creation.
- **NextForm** changes files into different formats, including: LDIF and CSV, MFVL and XML, MF-ISAM and Vision, structured and unstructured text.
- **NextForm** translates data types, including: EBCDIC to ASCII, packed decimal to numeric, ISO to European timestamp, Big5 to Unicode, etc.
- NextForm can also create multiple targets for replication or federation, and reformat and validate field and record layouts.

Appendix

A Data Types

This section lists the field data types that **FieldShield** supports natively.

Generally, both a form and type are required. There are five basic forms of data that **FieldShield** recognizes, shown in *Table 20*:

Table 20: Forms

Form	Name
0	Alphabetic
1	Numeric
2	Date
3	Time
4	Timestamp

In **FieldShield**, the data type's name is declared in **FieldShield** field statements, for example:

```
/FIELD=(Price,POSITION=35,SIZE=4,INTEGER)
```

The more than 100 internally supported data types meet the needs of most **FieldShield** users. **FieldShield** also has the ability to support special data types, such as encrypted data, multi-byte character sets, and ad hoc forms. This is done through user-written procedures (see the Custom Procedures *chapter on page 187* for details).

A.1 Form 0 -- Alphabetic Data Types

Form 0 types include ASCII, EBCDIC, and *multi-byte character* strings and a special date form.

Table 21: Single-Byte Types of Form 0

FieldShield Reference Type	Reference Standard
`ASCII` `[LATIN1]`	ISO 8859-1
`EBCDIC`	IBM Standard

Table 21: Single-Byte Types of Form 0

FieldShield Reference Type	Reference Standard
LATIN2	ISO 8859-2
LATIN3	ISO 8859-3
BALTIC	ISO 8859-4
CYRILLIC	ISO 8859-5
ARABIC	ISO 8859-6
GREEK	ISO 8859-7
HEBREW	ISO 8859-8
TURKISH [LATIN5]	ISO 8859-9
LATIN6	ISO 8859-10
FieldShield Reference Type	Examples
ASC_IN_EBC	"123">"ABC"
ASC_IN_NATURAL	"ABC">"A-D" (- is ignored)

The ASCII data type has two additional options: alignment and case folding. The alignment of ASCII characters in the field (none, left, or right) and their case sensitivity determine their exact collating sequence (see ASCII Collating Sequence *on page 239*).

Data types MONTH_DAY and ASC_in_EBC(ASCII in EBCDIC sequence) do not have the alignment and case folding options.

Table 22: ASCII Supplement

ASCII options	**FieldShield** Reference	Data Example
Not Aligned	ALIGNMENT NONE	" Chars "
Left Aligned	ALIGNMENT LEFT	"Chars "
Right Aligned	ALIGNMENT RIGHT	" Chars"
Case Sensitive	CASEFOLD YES	" Chars "
Not Sensitive	CASEFOLD NO	" CHARS "

Data types containing *multi-byte* characters are within Form 0 and Form 6. With all of the data types shown in *Table 23*, the sorting order is based strictly on the encoding order values.

For all Form 0 data types, API users should precede the *Reference Type* with `CS_`. For example, `KEF` should be `CS_KEF`.

Table 23: Multi-Byte Types of Form 0 (ASCII form)

sorti and **sortcl** Reference Type	Reference / Encoding Standard	Encoding Standard	Description / Comments
`JOHAB`	KS X 1001:1992	JOHAB	Korean multi-byte character separator unsupported
`KEF`	Korean EBCDIC Format	EBCDIC	Korean multi-byte character separator unsupported
`EHANGUL`	IBM DBCS-HOST	EBCDIC	Korean multi-byte character separator unsupported
`HHANGUL`	Hitachi Hangul		Korean multi-byte character separator unsupported
`UTF8 / UNICODE (Unicode)`	ISO 10646:1993-1	UTF8	Unicode
`UTF16 / UCS2 (Unicode)`	ISO10646	UTF16	Unicode
`UTF32 / UCS4 (Unicode)`	ISO10646	UTF32	Unicode
`GBK / EUC_CN`	ISO10646	GB13000	Chinese national standard
`BIG5`	BIG5		Hong Kong Chinese
`EUC_TW`	CNS 11643-1992 (Planes 1 - 3)	EUC	Taiwan
`EUC_KR`	KS X 1001:1992	EUC	Korean (WANSUNG)
`EUC_JP`	Plane 0: JIS X 0201-1976 Plane 1: JIS X 0208-1990 Plane 2: H/W Katakana Plane 3: JIS X 0212-1990	EUC	Japanese
`SJIS`	JIS X 0208-1990	Shift JIS	Japanese
`IBM_DBCS_HOST`	IBM Japanese IBM Korean IBM Simplified Chinese IBM Traditional Chinese	IBM DBCS HOST	IBM Simplified Chinese (mainframe encoding)
`IBM_DBCS_PC`	IBM Japanese IBM Korean IBM Simplified Chinese IBM Traditional Chinese	IBM DBCS PC	IBM Simplified Chinese (PC encoding)

Table 24 through *Table 27* show the multi-byte data types of form 6 recognized by **FieldShield**. All of the Unicode data types shown in these tables use double-byte ordering. The non-Unicode types support a mixture of 8-bit ASCII and double-byte native encodings. Sort orderings are noted in the *Collation Order* columns.

NOTE Dynamic libraries (.dll for Windows and .so for UNIX and Linux) are required for all form 6 data types in *Table 24* through *Table 27*. These libraries are provided at installation in the **$FIELDSHIELD_HOME/lib** directory.

Table 24: Chinese Big5 Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
`CHINESE_UNICODE_STROKE`	Bihua (Stroke) Unicode 5.2.0: http://www.unicode.org/ versions/Unicode5.2.0/	Chinese Unicode used in Taiwan, Hong Kong and Macao.	Includes Unicode characters, with Chinese characters sorted in stroke order. Includes CJK characters.
`CHINESE_BIG5,HK,MO, ROC,TW`	Bihua (Stroke) Windows Codepage 950: http://msdn.micro-soft.com/ en-us/goglobal/ cc305155.aspx	General, common usage BIG5.	Includes commonly used characters. Excludes punctuation, symbols, and rarely used characters.
		Commonly used Big5 in Hong Kong.	
		Commonly used Big5 in Macao.	
		Commonly used Big5 in the Republic of China.	
		Commonly used Big5 in Taiwan.	
`CHINESE_BIG5_RARE, HK_RARE,MO_RARE, ROC_RARE,TW_RARE`	Bihua (Stroke) Windows Codepage 950: http://msdn.micro-soft.com/ en-us/goglobal/ cc305155.aspx	General, rare usage BIG5.	Includes rarely used characters. Excludes punctuation, symbols, and commonly used characters.
		Rarely used Big5 in Hong Kong.	
		Rarely used Big5 in Macao.	
		Rarely used Big5 in the Republic of China.	
		Rarely used Big5 in Taiwan.	
`CHINESE_BIG5_DIGITS`		Digits.	Includes 10 digit characters.

Table 25: Chinese GBK Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
`CHINESE_UNICODE_PINYIN`	Pinyin Unicode 5.2.0: http://www.unicode.org/versions/Unicode5.2.0/	Chinese Unicode used in mainland, Singapore.	Includes Unicode characters, with Chinese characters sorted in Pinyin order. Includes CJK characters.

`CHINESE_GBK_SIMPLIFIED,`
`PRC_SIMPLIFIED,`
`SG_SIMPLIFIED,`
	Pinyin
	
	Windows Codepage 936: http://msdn.microsoft.com/en-us/goglobal/cc305153.aspx
		General, simplified GBK.	Includes simplified GBK.
			
			Excludes punctuation, symbols, commonly used traditional characters, and rarely used traditional characters.

		Simplified GBK used in the People's Republic of China.	
		Simplified GBK used in Singapore.	

`CHINESE_GBK_TRADITIONAL,`
`PRC_ TRADITIONAL,`
`SG_ TRADITIONAL`
	Pinyin
	
	Windows Codepage 936: http://msdn.microsoft.com/en-us/goglobal/cc305153.aspx
		General, common usage traditional GBK type.	Includes commonly used traditional characters.
			
			Excludes punctuation, symbols, simplified characters and rarely used traditional characters.

		Commonly used traditional GBK used in the Republic of China.	
		Commonly used traditional GBK used in Singapore.	

`CHINESE_GBK_TRADITIONAL_RARE,`
`PRC_ TRADITIONAL_RARE,`
`SG_ TRADITIONAL_RARE`
	Pinyin
	
	Windows Codepage 936: http://msdn.microsoft.com/en-us/goglobal/cc305153.aspx
		General, rare usage traditional GBK.	Includes rarely used traditional characters.
			
			Excludes punctuation, symbols, simplified characters and commonly used traditional characters.

		Rarely used traditional GBK used in the Republic of China.	
		Rarely used traditional GBK used in Singapore.	
`CHINESE_GBK_DIGITS`		Digits.	Includes 10 digit characters.

Table 26: Japanese Shift_JIS Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes

`JAPANESE_ALPHABET,`
`JP_ALPHABET`
	Dictionary order
	
	Windows Codepage 932: http://msdn.micro-soft.com/en-us/goglobal/cc305152.aspx
		Includes 56 Japanese alphabetic letters.	Includes Shift_JIS and half (ANSCII) characters.

`JAPANESE_HIRAGANA_BIG,`
`JP_HIRAGANA_BIG`
		Hiragana upper-case. Includes 12 Hiragana upper-case characters.	

`JAPANESE_HIRAGANA_SMALL,`
`JP_HIRAGANA_SMALL`
		Hiragana lower-case. Includes 71 Hiragana lower-case characters.	

`JAPANESE_HIRAGANA,`
`JP_HIGRAGANA`
		Hiragana upper-and lower-case. Includes 83 Hiragana characters.	

`JAPANESE_KATAKANA_FULL_BIG,`
`JP_KATAKANA_FULL_BIG`
		Katakana upper-case. Includes 12 Katakana upper-case characters.	

`JAPANESE_KATAKANA_FULL_SMALL,`
`JP_KATAKANA_FULL_SMALL`
		Katakana lower-case. Includes 71 Katakana lower-case characters.	

`JAPANESE_KATAKANA_FULL,`
`JP_KATAKANA_FULL`
		Katakana upper- and lower-case. Includes 83 Katakana characters.	

`JAPANESE_KATAKANA_HALF,`
`JP_KATAKANA_HALF`
		Katakana (single-byte). Includes 55 Katakana single-byte characters.	

`JAPANESE_KANJI,`
`JP_KANJI`
		Common usage Kanji. Includes 2,965 commonly used Kanji characters.	

`JAPANESE_KANJI_RARE,`
`JP_KANJI_RARE`
		Rarely used Kanji. Includes 3,390 rarely used Kanji characters.	

`JAPANESE_DIGITS,`
`JP_DIGITS`
		Digits.	Includes 10 digit characters.

Table 26: Japanese Shift_JIS Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
`JAPANESE_UNICODE_ALPHABETIC,` `JP_UNICODE_ALPHABETIC`	Dictionary order Unicode 5.2.0: http://www.unicode.org/versions/Unicode5.2.0/	Includes 52 Japanese Unicode alphabetic letters.	Includes Unicode characters, with Japanese characters sorted in dictionary order.
`JAPANESE_UNICODE_HIRAGANA_BIG,` `JP_UNICODE_HIRAGANA_BIG`		Unicode Hiragana upper-case. Includes 71 Hiragana upper- case characters.	
`JAPANESE_UNICODE_HIRAGANA_SMALL,` `JP_UNICODE_HIRAGANA_SMALL`		Unicode Hiragana lower-case. Includes 12 Hiragana lower-case characters.	
`JAPANESE_UNICODE_HIRAGANA,` `JP_UNICODE_HIRAGANA`		Unicode Hiragana upper- and lower-case. Includes 83 Hiragana characters	
`JAPANESE_UNICODE_KATAKANA _FULL_BIG,` `JP_UNICODE_KATAKANA _FULL_BIG`		Unicode Katakana upper-case. Includes 12 Katakana upper-case characters.	
`JAPANESE_UNICODE_KATAKANA _FULL_SMALL,` `JP_UNICODE_KATAKANA _FULL_SMALL`		Unicode Katakana lower-case. Includes 71 Katakana lower-case characters.	
`JAPANESE_UNICODE_KATAKANA _FULL,` `JP_UNICODE_KATAKANA _FULL`		Unicode Katakana upper- and lower-case. Includes 83 Katakana characters.	
`JAPANESE_UNICODE_KATAKANA_HALF,` `JP_UNICODE_KATAKANA_HALF`		Unicode Katakana (single-byte). Includes 55 Katakana single-byte characters.	
`JAPANESE_UNICODE_KANJI,` `JP_UNICODE_KANJI`		Unicode Kanji. Includes 2,965 commonly used Kanji characters.	

Table 27: Korean KSC5601 Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
`KOREAN_HANGUL,` `KR_HANGUL`	Dictionary order Windows Codepage 949: http://msdn.microsoft.com/en-us/goglobal/cc305154.aspx	Include 2,350 commonly used Hangul characters.	Includes KSC506 and ASCII characters.
`KOREAN_HANGUL_RARE,` `KR_HANGUL_RARE`		Includes 8,822 rarely used Hangul characters.	
`KOREAN _DIGITS,` `KR_DIGITS`		Digits.	Includes 10 digit characters.
`KOREAN_UNICODE_HANGUL,` `KR_UNICODE_HANGUL`	Dictionary order	Korean Unicode Hangul. Includes 11,172 Hangul characters.	Includes Unicode chars, with Korean characters sorted by Japanese dictionary order

Note that there is an additional data type, `UNCODE_DIGITS` that cannot be classified in any of the above language-specific tables. These are encodings in the range 0x0030 through 0x0039.

A.2 Form 1 -- The Numeric Types

If you select `NUMERIC` for form and `ASCNUM` for type, the data can be any combination of external numeric, human-readable characters. Numeric data can be used in calculations and can be classified as either external (human-readable) or internal.

External values can be a combination of signed or unsigned integers and reals. Within the specified field, leading spaces are ignored and a trailing space ends the quantity.

Examples of external numeric values are:

Integers	+123 -5 12345
Reals	-122.456 0.03 +12345.

Because these data types can be intermixed, no further specifications are necessary.

Table 28: Numeric Data Types

FieldShield Reference Type		Meaning
Alphanumeric		
0	`ASCNUM, NUMERIC`	Integers, reals, floating points
C Types		
1	`CHAR`	Character, Natural (ASCII)
2	`SCHAR`	Character, Signed
3	`UCHAR`	Character, Unsigned (EBCDIC)
4	`SHORT`	Integer, Short Signed
5	`USHORT`	Integer, Short Unsigned
6	`INT`	Integer, Natural Signed
7	`UINT`	Integer, Natural Unsigned
8	`LONG`	Integer, Long Signed
9	`ULONG`	Integer, Long Unsigned
10	`FLOAT`	Float, Single Precision
11	`DOUBLE`	Float, Double Precision

NOTE

For types 1-9 in *Table 28* on page 224, all **FieldShield** `/FIELD` declarations referencing these data types must contain `SIZE` attributes, even when the fields are delimited (see SIZE *on page 153*).

A.2.1 C Types

The first group of C types are supported by the C library in your computer's operating system. The fields are used by C, FORTRAN, Pascal and similar language applications.

Characters

Characters are one-machine-byte long, with no assumptions made about byte width for most comparison cases -- assuming a standard 8-bit width with ranges -128 ≤ signed character ≤ 127 and 0 ≤ unsigned character ≤ 255. *Natural* characters are either signed or unsigned, depending on the interpretation of `char` used by your machine.

1.	Character, Natural	1 ≤ Length
2.	Character, Signed	1 ≤ Length
3.	Character, Unsigned	1 ≤ Length

On machines without a signed character declaration, performance is degraded by forcing an additional interpretation. On most machines, characters are naturally signed, but `memcmp(a, b, 1)` will return the correct result when:

- 0 ≤ a ≤ 126 and 129 + a ≤ b ≤ 255
- 28 ≤ a ≤ 255 and 0 ≤ b ≤ a - 128

Integers and Floats

This category includes:

- multi-byte natural (single-byte ordering only)
- signed and unsigned short and long integers
- signed floats.

For many computers, a multi-byte field must be properly aligned on a memory address. This means that the field starting address must be an even multiple of ***n*** where ***n*** is the length of the key. Some operating systems sense misalignment and quickly and correctly shift the field. Most machines with RISC processors do not, however. **FieldShield** checks every compare for alignment, and applies correction when needed.

4.	Integer, Short Signed	2 ≤ Length.
5.	Integer, Short Unsigned	2 ≤ Length.

Short integers are stored as 2’s complement numbers (in binary form) usually in two 8-bit bytes with ranges -32,767 ≤ signed short ≤ 32,767 and 0 ≤ unsigned short ≤ 65535. No width assumptions are made. The order of the byte pair (whether higher valued bits are in the first or second byte) is the natural ordering of the hardware.

6.	Integer, Natural Signed	4 ≤ Length.
7.	Integer, Natural Unsigned	4 ≤ Length.

These two data types represent 2’s complement integers stored in the width allocated by an `int` declaration with no `short` or `long` adjective. Two and four bytes are common widths. Byte ordering is in the natural order of the machine.

8.	Integer, Long Signed	4 ≤ Length.
9.	Integer, Long Unsigned	4 ≤ Length.

Four bytes is the typical width for this 2’s complement data type, with ranges -2147483648 ≤ signed ≤ 2147483647 and 0 ≤ unsigned ≤ 4294967295.

10.	Float, Single Precision	4 ≤ Length.
11.	Float, Double Precision	8 ≤ Length.

Single Precision and Double Precision Floats are usually placed in four and eight bytes.

ASCII Numeric Data Types

The following are additional ASCII-numeric field data types that are supported by **FieldShield**:

- Currency
- IP Address
- Whole Number
- Bit

They will all, by default, display right-justified. `MONEY` and `CURRENCY` may be used interchangeably. They display the monetary symbol for the active *locale* at the beginning of the field.

The following are examples of how the ASCII-numeric data types display in the USA:

```
ASCII       Numeric    Currency Currency w/Fill

3.2            3.20       $3.20    $*******3.20
150          150.00     $150.00    $*****150.00
3.25           3.25       $3.25    $*******3.25
9.4562         9.46       $9.46    $*******9.46
1023.45     1023.56   $1,023.45    $***1,023.45
29384.56   29384.56  $29,384.56    $**29,384.56
```

`IP_ADDRESS` refers to positive whole number sub-fields separated by a dot (.), such as `155.46.142.205`.

`WHOLE_NUMBER` is an ASCII-numeric data type that displays right-justified and contains only integers.

`BIT` is used on input fields for which an ASCII or EBCDIC display of bit representation, e.g. 01110010, is required on output. The size of each field declared as `BIT` on input must be multiplied by eight for `ASCII` or `EBCDIC` output in order to display the complete bit representation.

Table 29: Types of Form 1

FieldShield Reference Type		Data Type
RM COBOL Types		
`12`	`RM_COMP`	COMP, Signed
`13`	`URM_COMP`	COMP, Unsigned
`14`	`RM_CMP1`	COMP-1
`15`	`RM_CMP3`	COMP-3, Signed
`16`	`URM_CMP3`	COMP-3, Unsigned
`17`	`RM_CMP6`	COMP-6
`18`	`RM_DISP`	DISP, Signed
`19`	`URM_DISP`	DISP, Unsigned
`20`	`RM_DISPSL`	DISP, Sign Leading
`21`	`RM_DISPSLS`	DISP, Sign Leading Separate
`22`	`RM_DISPST`	DISP, Sign Trailing
`23`	`RM_DISPSTS`	DISP, Sign Trailing Separate
Micro Focus COBOL Types		
`24`	`MF_COMP`	COMP, Signed
`25`	`UMF_COMP`	COMP, Unsigned
`26`	`MF_CMP3`	COMP-3, Packed Decimal
`27`	`UMF_CMP3`	COMP-3, Unsigned
`28`	`MF_CMP4`	COMP-4, Signed
`29`	`UMF_CMP4`	COMP-4, Unsigned
`30`	`MF_CMP5`	COMP-5, Signed
`31`	`UMF_CMP5`	COMP-5, Unsigned
`32`	`MF_CMPX`	COMP-X

Table 29: Types of Form 1 (cont.)

33	`MF_DISP`	DISP, Signed
34	`UMF_DISP`	DISP, Unsigned
35	`MF_DISPSL`	DISP, Sign Leading
36	`MF_DISPSLS`	DISP, Sign Leading Separate
37	`MF_DISPST`	DISP, Sign Trailing
38	`MF_DISPSTS`	DISP, Sign Trailing Separate

FieldShield Reference Type		Data Type
Miscellaneous		
39	`ZONED_DECIMAL`	Zoned Decimals
40	`ZONED_EBCDIC`	Zoned Decimals in EBCDIC
EBCDIC Native RM COBOL Types		
41	`ERM_COMP`	COMP, Signed_
42	`ERM_UCOMP`	COMP, Unsigned
43	`ERM_CMP1`	COMP-1
44	`ERM_CMP3`	COMP-3, Signed
45	`ERM_UCMP3`	COMP-3, Unsigned
46	`ERM_CMP6`	COMP-6
47	`ERM_DISP`	DISP, Signed
48	`ERM_UDISP`	DISP, Unsigned
49	`ERM_DISPSL`	DISP, Sign Leading
50	`ERM_DISPSLS`	DISP, Sign Leading Separate
51	`ERM_DISPST`	DISP, Sign Trailing
52	`ERM_DISPSTS`	DISP, Sign Trailing Separate
EBCDIC Native Micro Focus COBOL Types		
53	`EMF_COMP`	COMP, Signed
54	`EMF_UCOMP`	COMP, Unsigned
55	`EMF_CMP3`	COMP-3, Packed Decimal
56	`EMF_UCMP3`	COMP-3, Unsigned
57	`EMF_CMP4`	COMP-4, Signed
58	`EMF_UCMP4`	COMP-4, Unsigned
59	`EMF_CMP5`	COMP-5, Signed

Table 29: Types of Form 1 (cont.)

`60`	`EMF_UCMP5`	COMP-5, Unsigned
`61`	`EMF_COMPX`	COMP-X
`62`	`EMF_DISP`	DISP, Signed
`63`	`EMF_UDISP`	DISP, Unsigned
`64`	`EMF_DISPSL`	DISP, Sign Leading
`65`	`EMF_DISPSLS`	DISP, Sign Leading Separate
`66`	`EMF_DISPST`	DISP, Sign Trailing
`67`	`EMF_DISPSTS`	DISP, Sign Trailing Separate

A.2.2 RM COBOL Data Types

The following data types can be identified for comparisons.

12. RM COMP, Signed
13. RM COMP, Unsigned

`COMPUTATIONAL` variables are stored one digit per byte with a trailing byte for signed data. Each digit is stored as its binary value (that is, 0x01 for 1). The sign byte is 0x0D for negative values and 0x0B for positive values.

14. RM COMP-1

`COMPUTATIONAL-1` variables are stored as signed, two byte big-endian (most significant byte first) 2's complement numbers between -32,768 and 32,767.

15. RM COMP-3, Signed
16. RM COMP-3, Unsigned

A `COMPUTATIONAL-3` item, also known as *packed decimal*, is composed of a string of hex digits and a sign. Decimal digits (0 through 9) are held left to right. Each decimal digit is represented as a hex digit with two hex digits per byte.

The last hex digit holds the sign, as shown in *Table 29*:

Table 30: Hexadecimal/Signs

Decimal	Hex	Sign
11	b	Positive
13	d	Negative
15	f	Unsigned (positive)

The sign is the last hex digit so that an odd number or decimal digits needs to be retained. If there are an even number of digits, a 0 hex digit is prepended to the value to make full bytes.

Table 31 following are sample data representations:

Table 31: Sample Data Representations

Numeric Value	Hex Patterns	Bit Patterns
-123	12 3d	0001 0010 0011 1101
-1234	01 23 4d	0000 0001 0010 0011 0100 1101
+123	12 3b	0001 0010 0011 1100
+1234	01 23 4b	0000 0001 0010 0011 0100 1100

17. RM COMP-6, Unsigned

`COMPUTATIONAL-6` is stored just like `COMPUTATIONAL-3`, but as unsigned values without the need for a sign half-byte.

18. RM DISP, Signed
19. RM DISP, Unsigned
20. RM DISP, Sign leading
21. RM DISP, Sign leading separate
22. RM DISP, Sign trailing
23. RM DISP, Sign trailing separate

`USAGE IS DISPLAY` values are stored byte-by-byte as the ASCII values for each digit for up to 18 digits. Each digit is in the range 0x30 through 0x39.

`SIGN IS SEPARATE` causes a leading or trailing byte to be added to the width, with a value of 0x2B (ASCII ’+’) or 0x2D (ASCII ’-’).

Included signs cause the leading or trailing byte to be incremented by 16 for positive values if the digit is `1` through `9`, to `A` (0x41) through `I` (0x49), or to (0x7B) for `0`. Negative values increment `1` through `9` to `J` (0x4A) through `R` (0x52), and `0` to `}` (0x7D).

A.2.3 MF COBOL Data Types

These Micro Focus data types can be compared by **FieldShield**. The width of data fields is generally based on a maximum of 18 characters in a `PICTURE` clause.

24. MF COMP, Signed
25. MF COMP, Unsigned

`COMPUTATIONAL`, a.k.a. `COMPUTATIONAL-4` a.k.a. `BINARY`, negative values are stored as 2's complement numbers with the most significant byte first. The number of bytes of storage depends on the magnitude of the value (9s in PICTURE) and on the storage mode of the COBOL program which generates the data, as shown in *Table 32*:

Table 32: Picture 9s/Storage

Picture 9's		Storage	
Signed	Unsigned	Byte	Word
1-2	1-2	1	2
3-4	3-4	2	2
5-6	5-7	3	4
7-9	8-9	4	4
10-11	10-12	5	8
12-14	13-14	6	8
15-16	15-16	7	8
17-18	17-18	8	8

26. MF COMP-3, Signed
27. MF COMP-3, Unsigned
28. MF COMP-4, Signed
29. MF COMP-4, Unsigned

Micro Focus `COMPUTATIONAL-3`, packed decimal, is like Ryan-McFarland (RM/COBOL) `COMPUTATIONAL-3` but with 0xC used for the positive values sign in the half-byte instead of 0xB (see *Table 30* on page 230).

30. MF COMP-5, Signed
31. MF COMP-5, Unsigned

`COMPUTATIONAL-5` is like `COMPUTATIONAL-4` but the byte order depends on the hardware.

NOTE For **FieldShield** purposes, if you are using big-endian data you should use `COMP-4`. The `COMP-5` algorithm explicitly does little-endian comparisons.

32. MF COMP-X

`COMPUTATIONAL-X` is like `COMP-4`, but its width is the number of 9s in the `PICTURE` clause and its allowable values are any unsigned integer which fits the width, i.e., 41 significant decimal digits at most.

33. MF DISP, Signed
34. MF DISP, Unsigned
35. MF DISP, Sign leading
36. MF DISP, Sign leading separate
37. MF DISP, Sign trailing
38. MF DISP, Sign trailing separate

Micro Focus `DISPLAY` values are stored in the same manner as Ryan-McFarland, except with sign included. MF does not alter any bytes when a positive sign is included, and increments by 64, from `0` through `9` (0x30-0x39) to `p` through `y` (0x70-0x79).

39. ZONED, Zoned Decimal
40. ZONED_EBCDIC, Zoned Decimals in EBCDIC

Zoned Decimals are alphanumeric digits. If the decimal quantity is negative, the last character is zoned, i.e., written as a lower-case character. Only integers are represented and only integer values are recognized. **FieldShield** reads upper- and lower- case characters and writes lower case.

If the quantity is negative and:

Table 33:

the last character is	the string ends with
0	p
1	q
. . .	. . .
9	y

Table 34 contains examples of 3 character strings:

Table 34: Decimal/Zoned Decimal

Decimal	Zoned Decimal
1	001
-1	00q
10	010
-10	01p
999	999
-999	99y

NOTE In order to pad a zoned decimal output field with zeroes to the `SIZE` given, you must include the attribute `FILL='0'` (see FILL *on page 158*).

A.2.4 EBCDIC Native RM COBOL Data Types

41 - 52 see RM COBOL Data Types *on page 229* for descriptions.

A.2.5 EBCDIC Native MF COBOL Data Types

53 - 67 see MF COBOL Data Types *on page 231* for descriptions.

A.3 Forms 2-4 -- Date/Time/Timestamp Data Types

FieldShield recognizes the four distinct ways of describing time, date, and timestamp: American, European, Japanese, and International Standards Organization (ISO).

In *Table 35*, the example column illustrates the different forms possible for input entries. Output values, however, are always in the form of the last example in the table cells.

Table 35: Date/Time/Timestamp Data Types

FieldShield Name	Syntax
`AMERICAN_DATE`	*month*(name or integer)/*day*/*year*
`AMERICAN_TIME`	*hour*[:*minute*][:*second*] ***x***M
`AMERICAN_TIMESTAMP`	*month*/*day*/*year hour*[:*minute*][:*second*]
`EUROPEAN_DATE`	*day*.*month*(name or integer).*year*
`EUROPEAN_TIME`	*hour*[.*minute*][.*second*]
`EUROPEAN_TIMESTAMP`	*day*.*month*.*year hour*[.*minute*][.*second*]
`JAPANESE_DATE`	*year*-*month*(name or integer)-*day*
`JAPANESE_TIME`	*hour*[:*minute*][:*second*] ***x***M
`JAPANESE_TIMESTAMP`	*year*-*month*-*day hour*[:*minute*][:*second*]
`ISO_DATE`	*year*-*month*(name or integer)-*day*
`ISO_TIME`	*hour*[:*minute*][:*second*]
`ISO_TIMESTAMP`	*year*-*month*-*day hour*[:*minute*][:*second*]
`MONTH_DAY`	`"Jan"<"Feb"` and `"Wed"<"Thu"`

B Error Values

The errors and messages in *Table 36*, presented in value order, can occur during **FieldShield** execution. When an error occurs, batch programs stop with an error message, the system status error is set, and work files are purged. Depending on how your operating system was generated, fatal job or operator intervention errors may not be caught by **FieldShield** and may go on to cause undesirable results. In these cases, contact your IRI agent for assistance.

Table 36: Error Values

Value	Message	Meaning
0	Normal Return	N/A
2	Insufficient Memory	Required memory space could not be dynamically allocated.
3	Unknown Exception	An internal error detected due to invalid data and/or specifications. Check whether the input data and specifications are valid. Contact IRI Support if you cannot resolve the problem.
5	File Creation Error	A given file cannot be created. Make sure you have access to the directory and that the directory has enough room to create the file.
10	Parameter Error	Indicates that a routine was called with an illegal parameter. Check your **FieldShield** specification.
11	Record Length Improper	A record length less than 0 was specified.
23	Output Type Unknown	Output was not stdout, file, both, or returned to caller.
24	Problem with User's Output File	An error occurred when writing data to the final output file.
28	Environment Variable Undefined	Undefined environment variable.
29	Source Unknown	**FieldShield** syntax error.
30	Unexpected	**FieldShield** syntax error.
31	Wrong Combination of Items	**FieldShield** syntax error.
33	Terminated	Program execution terminated by Ctrl-C.
34	Insufficient Disk Space for Output File	Total input bytes cannot fit on disk where output file is specified.
46	License Violation: Incorrect Node or Invalid Key	Contact your IRI agent.
47	License Violation: Expiration Date Passed or Invalid Key	Contact your IRI agent.
48	License Violation: Invalid Key	Contact your IRI agent.
49	License Error: Cannot Obtain Machine ID.	License manager is unable to identify its location. Contact your IRI agent.

Table 36: Error Values (cont.)

Value	Message	Meaning
52	FIELDSHIELD_HOME is Not Set in the Environment	Set the environment variable `FIELDSHIELD_HOME` to the **FieldShield** home directory.
54	License Violation: This Application is Not Licensed	Contact your IRI agent.
70	Too Many Errors -- Aborting	**FieldShield** script analyzer stops after 8 syntax errors.
71	Incomplete Command	Missing part(s) of a command.
72	Expecting Names	**FieldShield** syntax error.
73	Parenthesis Count	**FieldShield** syntax error.
74	Missing Double Quote Mark	**FieldShield** syntax error.
75	Duplicate Name	**FieldShield** syntax error.
76	Expecting	**FieldShield** syntax error.
77	Expression Syntax	**FieldShield** syntax error.
78	Source is Elsewhere	**FieldShield** syntax error.
80	Field Length > Record Length	**FieldShield** syntax error.
81	Overlapping Field	**FieldShield** syntax error.
82	Illegal Character	**FieldShield** syntax error.
84	Unrecognized Word	**FieldShield** syntax error.
85	Scripts Too Deeply Nested	**FieldShield** syntax error.
86	Circular Definition	**FieldShield** syntax error.
87	Not an Active File	**FieldShield** syntax error.
88	Blocking Factor Invalid	**FieldShield** syntax error.
89	No File with this Name	**FieldShield** syntax error.
90	Unrecognized Name in Expression	Field referenced was not specified.
92	Not a Valid Option Here	Improper syntax.
95	Improper Command	Not a recognized command.
96	No Such Locale on System	Unknown locale specified.
97	Cannot Set Locale on System	The requested locale routines are not available.
98	Specific Line Too Long	**FieldShield** statement or command line limit exceeded. 16,383 bytes is the maximum line length.
101	Ambiguous Reference	**FieldShield** syntax error.
102	Invalid Record Length for this File	e.g., file size not integer multiple of fixed record length.
109	Invalid Conversion	Cannot convert this data type.
119	Unable to Read File	Error in reading the specified file.

Table 36: Error Values (cont.)

Value	Message	Meaning
120	Unable to Write File	Error in writing the specified file.
121	Unable to Open File	Error in opening the specified file.
123	File write permission denied	Insufficient privileges to write to the specified file
124	No Such File	The specified file does not exist.
125	Invalid File	The specified file is not a regular file.
126	Empty Input File	The specified file is empty.
128	Not Supported	The feature/syntax is not supported by **FieldShield**. Contact your IRI agent.
130	Missing records (number read != number written)	A mismatch between the number of output records and the number of input records.
151	Operating System Error	Indicates an operating system error that is not otherwise covered by the one of the standard error conditions.
154	Unsupported Action for the Current File Mode	An operation was requested that the current file open mode does not allow.
155	Record in Use by Another Process/Task	The requested record is locked by another process/task.
158	Requested Record Was Not Found	The requested record was not found. This can indicate the end or beginning of the file.
159	File Handler has Undefined Status	The current file operation cannot be completed because it detected an *undefined* status for a parameter.
160	Disk Full	Insufficient space for input/output files.
161	File in Use by Another Process/Task	The file is locked by another process/task.
162	Mismatch in Record Size	Mismatch detected in the record size specifications.
163	File Type Mismatch	Trying to treat a file with a different `/PROCESS` type.
166	Permission Denied	Insufficient privileges to access the specified file.
167	Requested Operation Not Supported by this Host System	The requested operation is not supported in your machine.
168	System Ran Out of Lock-Table Entries	Indicates an error when your machine ran out of lock-table entries. Try again.
169	Vision License Error	Invalid license file to access vision files. Make sure you have the Vision license file **FieldShield.vlc** in the directory where you have the **FieldShield** executable. The Vision license file can be obtained from an Micro Focus representative.
171	Error in the Vision Transaction System	Indicates that an error occurred in the Acucobol Vision transaction system.
172	Header information missing in input file	The input `/PROCESS` type you have specified requires a header record to exist, and it can not be found.

Table 36: Error Values (cont.)

Value	Message	Meaning
173	File read permission denied	Insufficient privileges to read from to the specified file.
178	[custom message]	This can vary depending on the nature of the error. Read the message for details.

C ASCII Collating Sequence

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
000	00	Nul	032	20	Bl	064	40	@	096	60	’
001	01	Soh	033	21	!	065	41	A	097	61	a
002	02	Stx	034	22	"	066	42	B	098	62	b
003	03	Etx	035	23	#	067	43	C	099	63	c
004	04	Eot	036	24	$	068	44	D	100	64	d
005	05	Enq	037	25	%	069	45	E	101	65	e
006	06	Ack	038	26	&	070	46	F	102	66	f
007	07	Bel	039	27	’	071	47	G	103	67	g
008	08	Bs	040	28	(	072	48	H	104	68	h
009	09	Ht	041	29	)	073	49	I	105	69	i
010	0A	Lf	042	2A	*	074	4A	J	106	6A	j
011	0B	Vt	043	2B	+	075	4B	K	107	6B	k
012	0C	Ff	044	2C	,	076	4C	L	108	6C	l
013	0D	Cr	045	2D	–	077	4D	M	109	6D	m
014	0E	So	046	2E	.	078	4E	N	110	6E	n
015	0F	Si	047	2F	/	079	4F	O	111	6F	o
016	10	Dle	048	30	0	080	50	P	112	70	p
017	11	Dc1	049	31	1	081	51	Q	113	71	q
018	12	Dc2	050	32	2	082	52	R	114	72	r
019	13	Dc3	051	33	3	083	53	S	115	73	s
020	14	Dc4	052	34	4	084	54	T	116	74	t
021	15	Nak	053	35	5	085	55	U	117	75	u
022	16	Syn	054	36	6	086	56	V	118	76	v
023	17	Etb	055	37	7	087	57	W	119	77	w
024	18	Can	056	38	8	088	58	X	120	78	x
025	19	Em	057	39	9	089	59	Y	121	79	y
026	1A	Sub	058	3A	:	090	5A	Z	122	7A	z
027	1B	Esc	059	3B	;	091	5B	[	123	7B	{
028	1C	Fs	060	3C	<	092	5C	\	124	7C	|
029	1D	Gs	061	3D	=	093	5D	]	125	7D	}
030	1E	Rs	062	3E	>	094	5E	^	126	7E	~
031	1F	Us	063	3F	?	095	5F	_	127	7F	Del

D EBCDIC Printing Characters

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
064	40	Bl				192	C0	{	240	F0	0
074	4A	¢	129	81	a	193	C1	A	241	F1	1
075	4B	.	130	82	b	194	C2	B	242	F2	2
076	4C	<	131	83	c	195	C3	C	243	F3	3
077	4D	(	132	84	d	196	C4	D	244	F4	4
078	4E	+	133	85	e	197	C5	E	245	F5	5
079	4F	|	134	86	f	198	C6	F	246	F6	6
080	50	&	135	87	g	199	C7	G	247	F7	7
090	5A	!	136	88	h	200	C8	H	248	F8	8
091	5B	$	137	89	i	201	C9	I	249	F9	9
092	5C	*				208	D0	}			
093	5D	)	145	91	j	209	D1	J			
094	5E	;	146	92	k	210	D2	K			
095	5F	^	147	93	l	211	D3	L			
096	60	-	148	94	m	212	D4	M			
097	61	/	149	95	n	213	D5	N			
106	6A	|	150	96	o	214	D6	O			
107	6B	,	151	97	p	215	D7	P			
108	6C	%	152	98	q	216	D8	Q			
109	6D	_	153	99	r	217	D9	R			
110	6E	>	161	A1	~	224	E0	\			
111	6F	?	162	A2	s	226	E2	S			
121	79	`	163	A3	t	227	E3	T			
122	7A	:	164	A4	u	228	E4	U			
123	7B	#	165	A5	v	229	E5	V			
124	7C	@	166	A6	w	230	E6	W			
125	7D	’	167	A7	x	231	E7	X			
126	7E	=	168	A8	y	232	E8	Y			
127	7F	"	169	A9	z	233	E9	Z			

FieldShield Index

Symbols

!< 164
!= 164
!> 164
/APPEND 32, 177
/CREATE 32, 177
/DATA 75
    defined 74
/FIELD 146
/FOOTREAD 36
    defined 36
/FOOTREC 75
    defined 33
/HEADREAD 34
    defined 34
/HEADREC 75
    defined 32
/HEADSKIP
    defined 28
/HEADWRITE 34
    defined 34
/INCLUDE
    defined 24
/INCOLLECT 30
    defined 30
/INFILE 16, 139
/INFILES 140
/INPROCEDURE 141
/INSKIP 29
/JOBCOLLECT 31
/JOBSKIP 30
/LENGTH 168
/LOCALE 182
/OUTCOLLECT 31
/OUTFILE 142
/OUTPROCEDURE 143
/OUTSKIP 29
/PROCESS 167
/RECSPERPAGE 37
/ROUNDING 86
/SPECIFICATION 144
/TAILREAD
    defined 36
/TAILSKIP
    defined 29
/TAILWRITE
    defined 36
/UPDATE 178
/VERSION 15
\’ 76
\" 75
\\ 75
\0 76
\a 75
\b 75
\f 75
\n 34, 76
\r 34, 75
\t 75
\v 76
18
%% 33
%c 33
%d 33
%E 33
%e 33
%f 33
%G 33
%g 33
%i 33
%o 33
%s 33
%u 33
%X 33
%x 33
< 164
<= 164
== 164
> 164
>= 164
$ 17, 76

Numerics

2string functions
    non_alnum2s 80
    non_alpha2s 79
    non_ascii2s 80
    non_digit2s 80
    non_ebc_let2s 80

non_print2s 79

A

abbreviations 17
abs (x) 86
accepting records 30
ACUCOBOL-GT file format 171
addition 85
alert 75
alignment of fields 157
AMERICAN_DATE 76
AMERICAN_TIME 76
AMERICAN_TIMESTAMP 76
appending to an output file 32, 177
arithmetic symbols
  -, subtraction 85
  (), parentheses 85
  *, multiplication 85
  +, addition 85
  unary 85
auditing 121

B

backslash 75
backspace 75
binary logical expressions 163
binary null 158
BLOCKED file format 180
building conditions 166

C

carriage return 75
case transfer functions
  tolower 80
  toproper 80
  toupper 80
case-sensitive 17
ceil (x) 86
CLF (NCSA Common Log Format) 182
columns. See fields, POSITION.
command line 195
comments 18
Common Log Format. See CLF.
compound conditions 166
compound logical expressions 164
compressed input files 141
compressed output files 142
conditional /DATA statements
  IF 20
  IF-THEN-ELSE logic 20
  THEN 20
conditional /FIELD statements
  IF 20
  IF-THEN-ELSE logic 20
  THEN 20
conditions
  binary 163
  compound 164, 166
  explicit (named) 26, 163
  implicit 163
  syntax 163
  WHERE 24, 163
    with include-omit 24
control characters
  \’, single quote 76
  \", double quote 75
  \\, backslash 75
  \0, null 76
  \a, alert 75
  \b, backspace 75
  \f, form feed 75
  \n, newline 76
  \r, carriage return 75
  \t, horizontal tab 75
  \v, vertical tab 76
  $, dollar sign 76
creating a new output file 32
creating an output file 177
CSV file format 172
csv2ddf 172
CT 164
CURRENCY 226
CURRENT_DATE 76
CURRENT_TIME 76
CURRENT_TIMESTAMP 76
CURRENT_TIMEZONE 76

D

data definition file 143
data files 139
Data masking 57
data type
    CURRENCY 226
    IP_ADDRESS 227
    NUMERIC 226
    WHOLE_NUMBER 227
data type conversion 159
data types 159
    dates 76
    multi-byte
        multi-byte data types 159
    timestamps 76
database extraction 176
database loading 177
dates 76
decimal point 153
decimal precsion of numeric fields 154
defined 31, 32
delimiter 150
different separators 152
dollar sign 76
double quote 75
DSN 174

E

ELF file format 181
elf2ddf 181
ELSE 19
environment variables 17, 145
EQ 164
EUROPEAN_DATE 76
EUROPEAN_TIME 76
EUROPEAN_TIMESTAMP 76
evaluation of include-omit 24
evaluation order of expressions 166
explicit conditions 26, 163
expressions
    binary logical 163
    compound logical 164
Extended Log Format. See ELF.
External values 224
external values 224
extraction 176

F

Fast Access Table (FAT) 17
field remapping 142
fields
    alignment 157
    data types 159
    FILL 158
    FRAME 155
    naming 147
    padding 158
    POSITION 148
    PRECISION 154
    reducing 158
    SEPARATOR 150
    SIZE 153
        precision 153
    trimming 158
FieldShield tools
    elf2ddf 181
file formats
    ACUCOBOL-GT 171
    BLOCKED 180
    CSV 172
    ELF 181
    LDIF 172
    LINE_SEQUENTIAL 170
    MFISAM 171
    MFVL_LARGE 170
    MFVL_SMALL 169
    ODBC 50, 173
    RECORD 168
    UNIVBF 171
    VARIABLE_SEQUENTIAL 170
    VSAM 171
    XML 180
files
    data 139
    data definition 143
    job specification 144
    output 142
    specification 144
FILL 158
fill characters in fields 158
find and replace

  case transfer functions 80
fixed-length records 168
floating-point precision 85
floor (x) 86
footer record 33
format control characters 33
  %%, to write % to output 33
  %c, character 33
  %d, %i, decimal integer 33
  %e, %E, %f, floating point number 33
  %G, E-format or F-format 33
  %g, e-format or f-format 33
  %o, unsigned octal integer 33
  %s, as a string 33
  %u, unsigned decimal integer 33
  %x, %X, unsigned hexadecimal integer 33
format_strings 59
form-feed 75
FRAME 155
framed characters in fields 155
framing 155

G

GE 164
globally unique identifiers. See GUID.
GT 164
GUID 61
gzip 141, 142

H

header record 28, 32, 34
horizontal tab 75
HTML 34

I

IF 20
IF-THEN-ELSE logic 19, 20
implicit conditions 163
include-omit 24
  evaluation 24
  examples 25
integers 224
internal variables
  AMERICAN_DATE 76
  AMERICAN_TIME 76
  AMERICAN_TIMESTAMP 76
  CURRENT_DATE 76
  CURRENT_TIME 76
  CURRENT_TIMESTAMP 76
  CURRENT_TIMEZONE 76
  EUROPEAN_DATE 76
  EUROPEAN_TIME 76
  EUROPEAN_TIMESTAMP 76
  ISO_DATE 76
  ISO_TIME 76
  ISO_TIMESTAMP 76
  JAPANESE_DATE 76
  JAPANESE_TIME 76
  JAPANESE_TIMESTAMP 76
  PAGE_NUMBER 77
  SYSDATE 76
  USER 77
  VALUE_TIME 76
  VALUE_TIMESTAMP 76
IP_ADDRESS 227
iscompares
  ispattern 165
ISO_DATE 76
ISO_TIME 76
ISO_TIMESTAMP 76

J

JAPANESE_DATE 76
JAPANESE_TIME 76
JAPANESE_TIMESTAMP 76
job specification file 144

L

language environment 182
LDIF file format 172
LE 164
leading space 224
left align 157
length 168
length (n) 86

letter case transfer 80
LINES 38
LINE_SEQUENTIAL file format 170
loading 177
locale 182
logical expressions 162
    binary 163
    compound 164
    evaluation order 166
lowercase 17
LT 164

M

mathematical functions 74, 86
    abs (x) 86
    ceil (x) 86
    floor (x) 86
    length (n) 86
    mod (x,y) 86
    pow (x,y) 86
MFISAM file format 171
MFVL_LARGE file format 170
MFVL_SMALL file format 169
mod (x,y) 86
multiplication 85

N

named conditions 26, 163
naming fields 147
NC 164
NCSA Common Log Format 182
NE 164
newline 76
nextform tools
    csv2ddf 172
NT File System. See NTFS.
NTFS 17
null 76, 158
NUMERIC 226
Numeric 227
numeric data 153

O

ODBC file format 50, 173
OMIT
    defined 24
optional statements 17
optional values 17
output files 142

P

padding fields 158
PAGE_NUMBER 34, 77
parentheses in arithmetic expressions 85
pattern matching 165
POSITION 148
pow (x,y) 86
PRECISION 154
precision 153
processing records 30

R

reals 224
RECORD file format 168
records
    fixed-length 168
    footer 33
    format control characters 33
    header 32, 34
    length 168
    selection 24
    variable-length 169
RECORD_SEQUENTIAL file format 168
redaction 74
reducing fields 158
relational operators
    CT 164
    EQ, == 164
    GE, >=, !< 164
    GT, > 164
    LE, <=, !> 164
    list of 164
    LT, < 164
    NC 164
    NE, != 164

remapping 102, 142
replace characters
  replace_chars 58
right align 157
Rounding 86
rows. See records.
running count 147

S

script 144
Secure Hash Algorithm 56
SEPARATOR 150
separator characters in fields 150
SEQUENCER 147
SET files
  character 63
  create 63
  date 64
  defined 62
  multi-column 64
  numeric 63
  syntax 65
single quote 76
SIZE 153
size of fields 153
  precision 153
skipping records 29
specification file 144
ssl 38
statements
  /APPEND 32, 177
  /CREATE 32, 177
  /DATA 75
    defined 74
  /FIELD 146
  /FOOTREAD 36
    defined 36
  /FOOTREC 75
    defined 33
  /HEADREAD 34
    defined 34
  /HEADREC 75
    defined 32
  /HEADSKIP
    defined 28
  /HEADWRITE 34
    defined 34
  /INCLUDE
    defined 24
  /INCOLLECT 30
    defined 30
  /INFILE 16, 139
  /INFILES 140
  /INPROCEDURE 141
  /INSKIP 29
  /JOBCOLLECT
    defined 31
  /JOBSKIP 30
  /LENGTH 168
  /LOCALE 182
  /OMIT
    defined 24
  /OUTCOLLECT
    defined 31
  /OUTFILE 142
  /OUTPROCEDURE 143
  /OUTSKIP 29
  /PROCESS 167
  /RECSPERPAGE
    defined 37
  /ROUNDING 86
  /SPECIFICATION 144
  /TAILREAD
    defined 36
  /TAILSKIP
    defined 29
  /TAILWRITE
    defined 36
  /UPDATE 178
  /VERSION 15
stdin 17
stdout 17
subtraction 85
SYSDATE 76

T

tail record 29
THEN 20
timestamps 76
Tokenization 53
tolower 80
toproper 80

toupper 80
trailing space 224
trimming fields 158

U

unary operators 85
UNIVBF file format 171
universally unique identifiers. See UUID.
updating an output file 178
uppercase 17
USER 34, 77
UUID 61

V

VALUE_TIME 76
VALUE_TIMESTAMP 76
variable-length records 169
VARIABLE_SEQUENTIAL file format 170
vertical tab 76
Vision (ACUCOBOL-GT) 171
VSAM file format 171

W

WHERE 24, 163
    with include-omit 24
WHOLE_NUMBER 227
wildcards (input file names) 141

X

XDEF 180
XML file format 180

We Need Your Input

IRI, Inc. upholds a standard of documentation quality that is best maintained through ongoing user feedback. If you have any comments, concerns, or suggestions regarding this documentation, please communicate them to us.

Please indicate the title and page number(s) in the manual you are addressing.
Be sure to provide your name, address (or e-mail address), and telephone number if you would like a reply from IRI.

Mailing address:	FieldShield Technical Publications Innovative Routines International, Inc. 2194 Highway A1A, Suite 303 Melbourne, FL 32937-4932 USA
Telephone:	USA +1 (321) 777-8889
Fax:	USA +1 (321) 777-8886
E-Mail:	support@iri.com

The comments you provide may be used by IRI to improve the quality of, or to make additions to, this document and/or the **FieldShield** software.