

# **GEDCOM to Fixed Format File Conversion Utility**

Prepared by the  
Family History Department  
The Church of Jesus Christ of Latter-day Saints

**6 January 2000**

## **Suggestions and Correspondence:**

E-mail:  
[gedcom@gedcom.org](mailto:gedcom@gedcom.org)

Telephone:  
801-240-4534

Mail:  
Family History Department  
GEDCOM Coordinator — 3T  
50 East North Temple Street  
Salt Lake City, UT 84150  
USA

## TABLE OF CONTENTS

- 1 Introduction**
  - 1.1 [What the GEDCOM to Fixed Format File Conversion Utility Does](#)
  - 1.2 [Conditions of Use](#)
  - 1.3 [Structure of this Document](#)
- 2 Overview**
  - 2.1 [Input and Output](#)
  - 2.2 [Control File Grammar](#)
  - 2.3 [File Definition Section](#)
- 3 Examples of Major Functions**
  - 3.1 [Executing the GEDCOM to Fixed Format File Conversion Utility](#)
  - 3.2 [Defining Output Records](#)
  - 3.3 [Initializing and Writing Output Records](#)
  - 3.4 [Putting Data in Output Records](#)
    - 3.4.1 [Basic Procedure](#)
    - 3.4.2 [Cross Reference ID's, Pointers, and Tags](#)
    - 3.4.3 [Multiple Occurring Tags](#)
  - 3.5 [Manipulating Output Data](#)
    - 3.5.1 [Reversing the Order of Sub-fields](#)
    - 3.5.2 [Using an Output Record as a Work Area](#)
    - 3.5.3 [Separating and Combining Sub-fields](#)
    - 3.5.4 [Another Method of Handling Multiple Occurring Tags](#)
  - 3.6 [Counting](#)
  - 3.7 [Selecting Records](#)
  - 3.8 [Printing Simple Reports](#)
- 4 Reference**
  - 4.1 [GEDCOM to Fixed Format File Conversion Utility Command Syntax](#)
  - 4.2 [General Directive Syntax](#)
  - 4.3 [Syntax of Directives \(in Alphabetic Order\)](#)
  - 4.4 [Error Codes](#)
- 5 Complete Examples**

# 1 Introduction

## 1.1 What the GEDCOM to Fixed Format File Conversion Utility Does

The GEDCOM to Fixed Format File Conversion Utility reads a file in the hierarchical structure defined in *The GEDCOM Standard*, and transfers selected data to one or more fixed format files. A "fixed format file" is a conventional file in which each record has the same format, defined by a fixed set of fields. Fixed format files can be used to import data into other applications (such as, database systems and spread sheets) or for direct processing (sorting, searching, displaying, etc.) The fixed format output files can contain fixed length fields, or delimited variable length fields. The utility can, also, produce simple reports listing data from the GEDCOM input file.

The major features of the utility are:

- C Conversion of data in GEDCOM format to conventional fixed format records, whose fields may contain tag values, tags, cross reference values, and pointers. The @ signs enclosing cross references and pointers are stripped.
- C Basic data editing. Subfields can be parsed into separate fields, fields can be concatenated in any order, with or without user specified delimiters, and alpha numeric dates can be converted to numeric dates. For example, "John H. /Smith/" can be edited to "Smith, John H." and "7 MAR 1935" to "1935/07/17"
- C Tag values from multiple occurring tags can be placed in a series of output fields or concatenated into a single variable length field.
- C Multiple counts can be kept of tag occurrences in the GEDCOM input file and numbers of records output.
- C Records can be selected for output based on counts or record content. Conditions can cause records to be either "included" or "excluded".
- C Basic reports, listing data from selected tags, with pagination, page headings, and column headings, can be created.

## 1.2 Conditions of Use

This program is made available in the public domain on an "as is" basis. THE CHURCH OF JESUS CHRIST OF LATTER-DAY SAINTS MAKES NO WARRANTIES WHETHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

### 1.3 Structure of this Document

The next (second) section of this document gives an overview of how the utility works, and the general syntax rules for writing the utility control code.

The third section shows, by example, how to code each major function of the utility. It is meant to quickly bring the user to an understanding of the utility, and enable him to begin using it.

The fourth section has the detailed syntax of each directive, in alphabetic order. It can be used for reference while coding.

The fifth section has complete examples of the utility control code so that each element can be seen in a complete, correct context.

## 2 Overview

### 2.1 Input and Output

The GEDCOM to Fixed Format File Conversion Utility reads a GEDCOM data file and a control file. It outputs fixed format files containing data extracted from the GEDCOM data file. The output files can contain fixed length or variable length fields, or can be a simple listing suitable for printing. The control file specifies what data is to be extracted from the GEDCOM data file, and the format and content of each flat output file.

### 2.2 Control File Grammar

The control file is written in GEDCOM grammar. That is, it consists of a hierarchy of tags and level numbers. ([Refer to the examples in section 5.](#)) The control file mirrors the tag and level number structure of the GEDCOM input data file. However, the control file does not contain data values. Instead it contains "directives" which specify actions to be taken each time a corresponding tag is found in the GEDCOM data. For example,

```
0 INDI
1 BIRT
2 DATE
3 - put_field type=a; field=4
```

directs the utility to write the value for each DATE tag (in the INDI, BIRT context) found in the GEDCOM input file to the field number 4 in output record "a". If the GEDCOM input file contains:

```

0 @I23@ INDI
1 NAME John /Doe/
1 SEX M
1 BIRT
2 DATE 17 MAR 1935
2 PLAC Ogden, Weber, UT, USA
1 DEAT
2 DATE 18 JUN 1992
...
0 @I24@ INDI
1 NAME ...
1 SEX F
1 BIRT
2 DATE 17 DEC 1934
...

```

the dates 17 MAR 1935 and 17 DEC 1934, would be written to two instances of record "a". (Code to write record "a" between each individual would be included in the control file, but is not shown in the example above.)

Notice the tags that mirror the GEDCOM input file are in uppercase. The match between the tags of the GEDCOM data file and the control file is case sensitive. The directives, however, are in lowercase to make them stand out.

A directive immediately follows the tag that triggers its function. Its level number must be exactly one higher than that of the associated tag.

The "-" in front of the directive in the above example, causes the directive to be executed immediately upon finding a DATE tag (in proper context) in the GEDCOM input file. A directive preceded with a "+" will be executed after all the subordinates of the associated tag are processed. For example, in

```

0 INDI
1 - init_record type=a
1 + put_record type=a
1 NAME
2 - put_field type=a; field=2
1 BIRT
2 DATE
3 - put_field type=a; field=4

```

record "a" is initialized as soon as an INDI tag is found in the GEDCOM input file. Similarly, the name and birth date are written to the record "a" buffer as soon as they are found. But record "a" is not written out (put\_record) until after all the subordinate tags of INDI are all processed (that is, until name and birth date are placed in the record buffer).

The control file does not have to contain all the tags in the GEDCOM data file. It need only specify the tags needed to find the data to be extracted. Of course, the context of tags used must be included. In the example above, the tag DATE cannot stand alone. The INDI and BIRT tags must be included to specify the context or meaning of DATE fully. The control file may contain tags and structures that do not occur in a particular GEDCOM data input file.

## 2.3 File Definition Section

The first section of the utility control file defines the output files. It does not mirror any structure or content of the GEDCOM input file. The following is an example of a file definition section [\(discussed in detail in Section 3.2\)](#):

```
0 DEFINITION
1 define_record type=a; file=c:\g2f\people.txt
2 define_field field=1; length=6; default="No ID"
2 define_field field=2; length=35; default="No Name"
2 define_field field=3; length=2
2 define_field field=4; length=12; justify=r
```

Notice, there are no - or + signs in front of the definition directives.



### 3 Examples of Major Functions

#### 3.1 Executing the GEDCOM to Fixed Format File Conversion Utility

The following is an example of the command to execute the GEDCOM to Fixed Format File Conversion Utility:

```
GED2FIX DATAIN.GED CONTROL.CFG OUTDATA.TXT
```

if the current directory is where GED2FIX is stored. Otherwise, specify the full path before GED2FIX.

Data is extracted from the GEDCOM data file, DATAIN.GED. The control file, CONTROL.CFG, specifies what data is to be extracted and the format of the output files. The OUTDATA.TXT defines the output path to where the new fixed formatted data will be written.

If a file name (file=) is not specified in any of the output file definitions, their output is sent to the default output file name specified on the command line. For example,

```
GED2FIX DATAIN.GED CONTROL.CFG C:\G2F\DATA\OUTDATA.TXT
```

directs the output to the file OUTDATA.TXT, and

```
GED2FIX DATAIN.GED CONTROL.CFG LPT1:
```

directs the output to a printer.

### 3.2 Defining Output Records

The following is an example of the file definition section of the control file ([from Section 5, Example 1](#)):

```
0 DEFINITION
1 define_record type=a
2 define_field field=1; length=3; default="No ID"
2 define_field field=2; length=20; default="Name Missing"
2 define_field field=3; length=2
2 define_field field=4; length=12; justify=r
2 define_field field=41;length=12; justify=r
2 define_field field=42;length=2
2 define_field field=5; length=20
1 define_record type=b; file=c:\g2f\families.txt; fdlim="|"
2 define_field field=1; length=0; default="Missing ID"
2 define_field field=2; length=0
2 define_field field=3; length=0
2 define_field field=4; length=0
2 define_field field=5; length=0
```

- C Two files are defined here, "a" and "b" (type parameter). Up to 5 files, "a" through "e", can be defined.
- C The first file, "a", has fixed length fields, the second, "b", has variable length fields. A value of 0 in the length parameter indicates a variable length field. Variable length fields are delimited by the value of the fdlim parameter (this can be a single character or a character string).
- C In variable length fields, the delimiter follows every field, including the last. This last delimiter at the end of a record will cause some programs to create an extra, empty field or column.
- C File "a" has no file name specified, so any records it outputs will be sent to the file name specified as the default "output file name" in the command line. The records of file "b" in this example will be written to the file  
families.txt  
This file name must not already exist. If it does, the run may fail.

There is a special option that can be used with the "file=" parameter in defining a record type. If the "file=" parameter is file="output.fam" then the output for this file type will be written to a file name generated by taking the same name as specified in the command line

for the default output file but changing the extension to the same extension given by the “file=” parameter. In this case it would provide the type b output with a “.FAM” extension. The input file name can be used in the same way by using the file=”input.IND” so that the output file would use the input default file name from the command line with the “.IND” file extension for this example.

- C The fields appear in an output record in the order they are defined in the DEFINITION section, not in the numeric order of the field parameter. In file "a", the order of the fields in the output record is 1, 2, 3, 4, 41, 42, 5.
- C The fields with "justify=r" are right justified. A "justify=l", or no justify parameter, causes a field to be left justified. See section 3.55 for other justification parameters.
- C For fixed length fields, the default fill character is a space. Another fill character can be specified. For example:

```
. . . justify=r; fill="0"
```

will right justify with leading zeros.

- C The values specified in the default parameter are placed in a record when it is initialized. They remain there unless overwritten by a put\_field or put\_xref directive.

Though they can be troublesome field delimiters, some programs require comma delimiters. In this case, if the data contains commas, the fields must be enclosed in quotes. The following example shows how to create a file of the form:

```
"Field1","field2","field3", . . .
```

```
1 define_record type=c; file=c:\g2f\families.txt; fdlim="",""  
2 define_field field=91;length=1; default=""  
2 define_field field=1; length=0; default="Missing ID"  
2 define_field field=2; length=0  
2 define_field field=3; length=0  
2 define_field field=92;length=1; default=""
```

- C In this example, the field delimiter is the string ",". Notice it is coded with two quotes, a comma, and two more quotes. There are 4 " values coded in the fdlim statement.
- C To get the leading quote for the first field, an extra one character fixed length field is defined before it (field 91), with a default value of ". Notice the default value is coded as 3

" values. Similarly, to get the trailing quote on the last field, a final fixed length field is added at the end (field 92) with a default value of ". (Each variable length field is followed by the delimiter. So without this last field, the end of the record would be

. . . "fieldx ","fieldy ","fieldz ","

The last quote has no closing quote. Adding the last fixed length field with a value of ", gives:

. . . "fieldx ","fieldy ","fieldz ",""

The last two quotes are now properly paired, but notice we have an extra, empty field.)

### 3.3 Initializing and Writing Output Records

Typical code for initializing and writing records is (from [Section 5, Example 1](#)):

```
0 INDI
1 - init_record type=a
1 + put_record type=a
1 NAME
2 - put_field type=a; field=2; . . .
1 BIRT
2 DATE
3 - put_field type=a; field=4
```

- C When an INDI tag is found in the input GEDCOM data file, record "a" is immediately initialized by the init\_record directive, since it is preceded by a "-" sign. This places the default values, if any were defined, in its fields. The default will remain in a field unless a put\_field or put\_xref replaces it.
- C The "+" sign before the put\_record causes it to be executed after all the tags subordinate to INDI are processed. In other words, the record is written out after an individual's name and birth date are put into the record buffer.
- C The init\_record directive, also, releases memory being held by record "a". It is possible to continually "overwrite" the field values and output records without initializing between writes. However, the "overwriting" of a field is logical. The memory already in use by a field is not reused nor released by "overwriting" the field. Repeated writes without initializing the record continually consumes memory. It can cause the utility to run out of memory and terminate.

### 3.4 Putting Data in Output Records

#### 3.4.1 Basic Procedure

The code ([from Section 5, Example 1](#)):

```
0 INDI
1 - init_record type=a
1 + put_record type=a
1 NAME
2 - put_field type=a; field=2; . . .
1 BIRT
2 DATE
3 - put_field type=a; field=4
```

also, shows the basics of putting field values in output record fields. Here an individual's name and birth date are placed in fields 1 and 2 of record type "a".

### 3.4.2 Cross Reference ID's, Pointers, and Tags

Consider the following from a GEDCOM data file:

```
0 @F23@ FAM
1 HUSB @I34@      (Pointer to Individual below, who is the father)
1 WIFE @I21@
1 MARR
2 DATE 10 APR 1873
2 PLAC Fullerton, Orange Co., Calif.
1 CHIL @I56@
1 CHIL @I57@
. . .
0 @I34@ INDI      (Individual record for the father, HUSB, in family above)
1 NAME . . .
1 FAMS @F23@      (Pointer to family above, in which this individual is a spouse)
```

On the first line, the @F23@ is the **cross reference** ID for this family record. It is used in records of individuals who belong to this family to connect to it (see last line of the code above). On the next line, the **tag** "HUSB" has a value @I34@. It is a **pointer** to the cross reference ID of the individual who is the father in this family (see INDI line in the code above).

The following code will preserve the relationships from the FAM records in the output file "d" (from [Section 5, Example 2](#)):

```
0 FAM
1 - init_record type=d
1 - put_xref type=d; xref=1
1 HUSB
2 - put_xref type=d; tag=2; ptr=3
2 - put_record type=d
1 WIFE
2 - put_xref type=d; tag=2; ptr=3
2 - put_record type=d
1 CHIL
2 - put_xref type=d; tag=2; ptr=3
2 - put_record type=d
```

The parameters xref=1, tag=2, and ptr=3 put the indicated values in fields 1, 2, and 3. The output from this is four records:

F23	HUSB	I34
F23	WIFE	I21
F23	CHIL	I56
F23	CHIL	I57

- C The family ID number (xref), the relationship (tag), and the individual's ID number (ptr) are preserved for each family member.
- C Since the directives for a tag are applied to each occurrence of the tag in the GEDCOM input, directives for the CHIL tag are applied to both instances of the CHIL tag in the input data, thus there are two CHIL records in the output above.
- C The @ signs in the pointers and cross references are stripped.
- C The put\_xref directive can write more than one field to the output record. (Up to 3 values, the cross reference ID, the tag, and the pointer.)
- C Record "d" was not initialized each time it was written. If it were, the family ID, F23, would appear only in the first record. The space used by each instance of a field is not released or reused between initializations, so repeatedly "writing over" the fields consumes memory. However, the record will be initialized for each family. Since a family has relatively few members, there is little danger of this causing a memory problem.

The following code accomplishes the same thing, but it initializes the output record between each put\_record.



## 0 DEFINITION

```
1 define_record type=d ; file=d:\g2f\fammembr.txt
2 define_field field=1; length=5
2 define_field field=2; length=6
2 define_field field=3; length=5
1 define_record type=e
2 define_field field=1; length=0
```

## 0 FAM

```
1 - init_record type=e
1 - put_xref type=e; xref=1
```

## 1 HUSB

```
2 - init_record type=d
2 - put_record type=d; field=1; ttype=e; fetch=1
2 - put_xref type=d; tag=2; ptr=3
2 - put_record type=d
```

## 1 WIFE

```
2 - init_record type=d
2 - put_record type=d; field=1; ttype=e; fetch=1
2 - put_xref type=d; tag=2; ptr=3
2 - put_record type=d
```

## 1 CHIL

```
2 - init_record type=d
2 - put_record type=d; field=1; ttype=e; fetch=1
2 - put_xref type=d; tag=2; ptr=3
2 - put_record type=d
```

- C This example, uses record "e" for a temporary storage area ([see section 3.5.2](#) for additional information). The common value, family ID, is saved in record "e". The line

```
put_field type=d; field=1; ttype=e; fetch=1
```

retrieves the family ID from field 1 of record "e" (fetch and ttype parameters) and writes it to field 1 of record "d".

- C This is a better way to handle a value used in many output records, because the memory used by the records is freed for reuse each time a record is written (since it is initialized before each write), rather than letting it accumulating to dangerous levels.

### 3.4.3 Multiple Occurring Tags

Some tags can occur multiple times within the same context. For example, a family may have any number of children, or text fields may have multiple continuation values. Consider the following GEDCOM data:

```
0 FAM
1 NOTE Information on this family was found
2 CONT in the family bible of John Smith,
2 CONT Native American verbal traditions,
2 CONT and the Encyclopedia Britannica
```

The following code uses a "multiple field group". Multiple output fields and a pointer are used to place each value of a repeating tag in successive output fields (from [Section 5, Example 1](#)):

```
0 FAM
1 - init_record type=b
1 - reset_multi_fld_grp group=1
1 + put_record type=b
1 NOTE
2 - put_multi_field type=b; group=1; field=6,7,8,9
2 CONT
3 - put_multi_field type=b; group=1; field=6,7,8,9
```

- C Each put\_multi\_field stores a value in the current field, and moves the field pointer to the next field in the list.
- C This code places the NOTE value and its CONT values in fields 6 through 9 of record "b". If there are more than four values for this note, those beyond the fourth will be lost.
- C Here multiple field group 1 is used. Up to 5 multiple field groups can be used concurrently.
- C Notice that the reset\_multi\_fld\_grp directive is used to reset the multiple field group pointer to the first field before each set of tags is processed.

Another way of handling the values from multiple occurring tag is explained in [Section 3.5.4](#).

## 3.5 Manipulating Output Data

The GEDCOM to Fixed Format File Conversion Utility can make some basic modifications to data before it is written out.

### 3.5.1 Reversing the Order of Sub-fields

Data such as dates, places, and names are generally composite values. For example, a date is a day, month, and year. The parts or sub-fields may be in the wrong order for your needs. For example, a date may be in day/month/year format, while to sort it needs to be in the order year/month/day.

To modify a data value, it is first read into an output field, and then modified. Suppose we want to sort our output file by locality. Since places are normally stored in smallest to largest locality (city, county, state, country) the order must be reversed for proper sorting.

If the GEDCOM input file contains:

```
0 @F23@ FAM
...
1 MARR
...
2 PLAC Fullerton, Orange Co., Calif., USA
...
```

the control file code (from [Section 5, Example 1](#)):

```
0 FAM
...
1 MARR
...
2 PLAC
3 - put_field type=b; field=51
3 - order_field type=b; field=51; parse=","; order=reverse
...
```

will result in field 51 containing

```
USA, Calif., Orange Co., Fullerton
```

Notice that although we are parsing on just the comma, the resulting sub-fields are separated by a comma and a space.

Dates present additional problems. If the date "4 MAR 1945" is reversed to "1945 MAR 4", it still will not sort properly. It must be changed to "1945 03 04". The following code accomplishes this (from [Section 5, Example 1](#)):

```
0 INDI
...
1 BIRT
1 DATE
2 - put_record type=a; field=41
2 - order_field type=a; field=41; parse=" "; order=reverse; date=numeric
...
```

Here we have parsed on a space. The "date=numeric" changes the month and day to the proper numeric format.

### 3.5.2 Using an Output Record as a Work Area

In the previous example of manipulating data, the data was read into its output field and modified in place. Sometimes that is not possible or convenient. The data can be read into another work field, modified, and then moved to the final output field(s). Fields used for temporary storage may be in an output record used exclusively for temporary storage (and is never written out), they can be fields in the current output record (which may be used later for other data), or they can be fields in another output record that is not being used at the moment. The safest approach is the first; use a separate record exclusively for temporary storage.

The following might be used to define a record for temporary storage ([from Section 5, Example 1](#)):

```
0 DEFINITION
...
1 define_record type=e
2 define_field field=1; length=0
2 define_field field=2; length=0
2 define_field field=3; length=0
...
```

- C Variable length fields are the most versatile approach for temporary storage.
- C Since the logical reuse of fields does not reuse or release memory, records used for temporary storage need to be initialized at appropriate times.

### 3.5.3 Separating and Combining Sub-fields

To illustrate these functions, suppose a name in the GEDCOM input file is in the form

John Henry /Doe/

and we want to change it to

Doe, John Henry.

We will read the field into a temporary storage area, split it into two fields, and then paste the pieces back together in the output record. The following code will do that (from [Section 5, Example 1](#)):

```
0 INDI
1 - init_record type=a
1 - init_record type=e
1 + put_record type=a
1 NAME
2 - put_field type=e; field=1
2 - order_field type=e; field=1; order=2,3; parse="/"
2 - put_field type=a; field=2; ttype=e; fetch=3
2 - put_field type=a; field=2; conc=", "; ttype=e; fetch=2
```

- C The line **put\_field type=e; field=1** puts the name in the original format into field 1 of record "e".
- C The next line, **order\_field type=e; field=1; order=2,3; parse="/"**, splits the sub-fields of the field 1 (full name) into two parts based on the delimiter "/". The first sub-field (given names) is placed in field 2, and the next sub-field (surname) is placed in field 3. If there were more than two sub-fields, more field numbers could have been specified in the "order=" parameter. The "order=" parameter could have used field 1 as the recipient of a sub-field, as well as the source of data to be separated.
- C The line **put\_field type=a; field=2; ttype=e; fetch=3** fetches the value in field 3 of record "e" (surname) and places it in field 2 of record "a".
- C Finally, **put\_field type=a; field=2; conc=", "; ttype=e; fetch=2** fetches field 2 from record "e" (given names) and concatenates it to the current contents of field 2 of record "a", preceded by the value in the "conc=" parameter; in this, case a comma and a blank.

Field 1 of record "a" now contains the full name in the "last, given names" format.

### 3.5.4 Another Method of Handling Multiple Occurring Tags

The "conc=" parameter can, also, be used to handle multiple occurring fields. For example (from [Section 5, Example 2](#)),

```
0 FAM
...
1 NOTE
2 - put_field type=c; field=6
3 CONT
4 - put_field type=c; field=6; conc=" "
```

will put all the note text and its continuations into field 6 with each value separated by a blank.

### 3.5.5 Using the Justification Code to convert to UPPERCASE or to change names of people or places to proper upper/lower case style.

The Justification parameter in the [DEFINE\\_FIELD](#) primitive can be used to convert the GEDCOM value to one of the following ways:

justify=r	will right justify the data.
justify=l	will suppress leading blanks or zeroes and left justify.
justify=u	will convert to upper case.
justify=p	will convert to lower case except for characters which follow blanks, commas, periods, or are at the beginning of a line. These are converted to upper case.

For example ([from Section 5, Example 2](#)),

```
0 define_record type=a
. . .
1 define\_field field=6; justify=p
1 define_field field=7; justify=u
```

This will cause any data placed into field 6 of record type a to be edited so that all characters are lower case except the first character of a field and those following a blank, a comma, or a slash. The last line will cause data placed in field 7 to be changed to upper case. [See Section 4.3](#) [define\\_field](#).



### 3.6 Counting

The GEDCOM to Fixed Format File Conversion Utility can keep multiple counts as it processes records. As an example, the following code counts the occurrences of FAM records, and stores the count in a summary record ([from Section 5, Example 3](#)):

```
0 HEAD
1 - reset_count counter=2; reset=0
0 FAM
1 - add_to_count counter=2; add=1
0 TRLR
1 - put_field field=6; type=b; fetch=count; counter=2
1 - put_record type=b
```

C Here counter 2 is being used. There are a maximum of 300 counters, 1 through 300.

C In reset\_count, the reset parameter sets an initial value for the counter.

C In the add\_to\_count, the add parameter is the amount to be added to the counter.

C The put\_field fetches the value in counter 2, and places it in field 6 of record "b".

The code below ([from Section 5, Example 3](#)) uses the put\_record to do the counting. Then the counter is incremented only when a record is actually written. If a select\_record is being used (see Section 3.7), the count will not be the number of INDI records found, but the number of records written.

```
0 HEAD
1 - reset_count counter=1; reset=0
0 INDI
1 - init_record type=c
1 - init_record type=e
1 + select_record type=c; ttype=e; fetch=2; include="twist"; when=eq
1 + put_record type=c; counter=1; add=1
1 NAME
2 - put_field type=c; field=2
2 - put_field type=e; field=1
2 - order_field type=e; field=1; order=1,2; parse="/"
0 TRLR
1 - put_field field=5; type=b; fetch=count; counter=1
1 - put_record type=b
```

The following code (from Section 5, Example 3) will sequence number the family output records and count the number of families (counter 2), and count the number of children in each family (counter 3).

```

0 HEAD
1 - reset_count counter=2; reset=0
0 FAM
1 - init_record type=d
1 - add_to_count counter=2; add=1          /* Increment family count & sequence #
1 - put_field type=d; field=1; fetch=count; counter=2      /* Put sequence number
1 - reset_count counter=3; reset=0
1 + put_field type=d; field=5; fetch=count; counter=3      /* Put number of children
1 + put_record type=d
1 CHIL
2 - add_to_count counter=3; add=1          /* Increment child count
0 TRLR
1 - put_field field=6; type=b; fetch=count; counter=2      /* Family count in summary
1 - put_record type=b

```

- C Counter 2 is reset only at the beginning of the run, keeping an ongoing count. Counter 3 is reset for each family so the count is the number of children within each family.

### 3.7 Selecting Records

There are two basic modes for selecting records:

1. A "write switch" can be turned on when a counter (or field value) reaches a specified value, and turned off at another value. All records are printed while the switch is on. This is useful for selecting a number of records for limited testing, sampling, etc.
2. Records are selected or rejected individually based on their satisfying specified conditions.

Selection of records is controlled separately for each record type.

The following example uses the first mode to select a range of family records (from Section 5, [Example 4](#)):

```
0 HEAD
1 - reset_count counter=2; reset=0
0 FAM
1 - init_record type=b
1 - add_to_count counter=2; add=1
1 + select_record type=b; counter=2; start=2; when=eq
1 + select_record type=b; counter=2; stop=4; when=eq
1 + put_record type=b
1 HUSB
...
```

- C The add\_to\_count directive keeps a count, in counter 2, of the number of FAM records which have been found.
- C The first select\_record causes the write flag for record type "b" to be turned on when the 2nd FAM record is found (start=2). The second turns the write flag off when the 4th FAM record is found (stop=4). Since the "stop" is processed before the put\_record, the 4th FAM will not be output. Families 2 through 3 will be written.
- C The select\_record directives are based on testing values of counters and fields updated during the processing of a record. They are, therefore, normally processed just before the associated put\_record directive, so that all processing of the record is completed before tests are performed.
- C The when=eq is the comparison operator for counter 2 and the 2 or 4. The "eq" stands for equal. When starting and stopping on a counter, equal is normally used. However, not equal, "ne", greater than, "gt", less than or equal, "le", etc. can be used if needed.

The following is an example of selecting individual records based a field value (from Section 5, [Example 3](#)):

```
0 INDI
1 - init_record type=c
1 - init_record type=e
1 + select_record type=c; ttype=e; fetch=2; include="twist"; when=eq
1 + put_record type=c; . . .
1 NAME
2 - put_field type=c; field=2
2 - put_field type=e; field=1
2 - order_field type=e; field=1; order=1,2; parse="/"
```

- C Here the individual's name is read into work record "e", and separated into given names and last name, with the last name in field 2 of record "e".
- C The select\_record controls the writing of record type "c" by fetching field 2 from record "e" (the last name) and comparing it to "twist". If it is equal, the record is "included", that is, it is written to the output file. The compare is not case sensitive. All individuals whose last name is Twist will be written out. The field used for the comparison can be in the record itself, rather than in a work area (ttype could be equal to "c").
- C Again, notice that the select\_record is processed just before the associated put\_record.
- C An "include" causes the record to be written. An "exclude" can be used to cause selected records to not be written.
- C Here equal, "eq", is used for comparison of the last name to "twist". "ne", "gt", "le", etc. could, also, be used.

The write switch set by the start/stop and the one set by include/exclude are separate, and work independently. If both types of select are in use, a record is written when both switches are on.

Multiple include and exclude statements can be used, and each will effect the include/exclude write switch in its turn when executed.

The following example shows several select statements acting together (from [Section 5, Example 4](#)):

```
0 HEAD
1 - reset_count counter=1; reset=1
0 INDI
1 - init_record type=a
1 - init_record type=e
1 + put_field type=a; field=1; fetch=count; counter=1
1 + select_record type=a; counter=1; start=1; when=eq
1 + select_record type=a; counter=1; stop=21; when=eq
1 + select_record type=a; ttype=e; fetch=2; include="g"; when=ge
1 + select_record type=a; ttype=e; fetch=2; exclude="u"; when=ge
1 + put_record type=a; counter=1; add=1
1 NAME
2 - put_field type=a; field=3
2 - put_field type=e; field=1
2 - order_field type=e; field=1; order=1,2; parse="/"
```

- C Counter 1 is incremented by the put\_record statement, that is, the count is of the number of records written, not the number of INDI records found. The output will be 20 records which satisfy the include/exclude conditions.
- C Notice the initial value of the counter for the start/stop switch is 1, and the start/stop selection is started at 1. Don't use another value. Since the counter is incremented by the put\_record, "start=10" would say "start writing records after 9 records have already been written." That doesn't work. Actually, the start/stop switch is on by default, so simply omitting the "select\_record start=" directive in this case will work properly and is less error prone.
- C Writing is stopped after 20 records are written. To be consistent with a previous example, it was coded "select\_record stop=21". This stops writing before the 21st record is written. In order to think in this fashion, and since counter 1 is incremented after the select\_record, it had to be initialized to 1. Initializing counter 1 to 0, and stopping on 20 would give the same results, and may seem more natural. (However, if a "start" is included, it would have to start at 0.)
- C The include/exclude mode of selection is controlled by a write switch that can be set by any number of successive "select\_record include/exclude" directives. The order of multiple include/exclude selection directives can be significant. The first include/exclude sets the switch honoring both explicit and implied conditions. For example,

```
select_record include="smith"; when=eq; . . .
```

will set the switch on for smith, and off for any other value. Subsequent include/excludes honor only the explicit condition, that is, include="jones" will set the switch on for jones, but will not set it off for other values; thus it does not override the selection of "smith" and both smiths and jones' will be written.

C The example above includes the code:

```
select_record include="g"; when=ge; . . .  
select_record exclude="u"; when=ge; . . .
```

The first statement turns the write switch on for all names starting with "g" through "z", and off for all others. The second turns the switch off for all names starting with "u" through "z" (but does not effect the switch for others). Thus the switch will be on for names starting with "g" through "t". If the order of these statements is reversed, the result is quite different. The

```
select_record exclude="u"; when=ge; . . .
```

would set the switch off for "u" through "z", and on for "a" through "t". The

```
select_record include="g"; when=ge; . . .
```

would then turn it on for all last names starting with "g" through "z". Since it is already on for "a" through "t", all records would be written.

### 3.8 Printing Simple Reports

To produce a simple report listing from data in the GEDCOM input data file, two output files are used. In the first, the default field values are used to define report headings. The second is used to output the actual data to be listed. In the example below, both are missing the file parameter, so the output is through the console data stream (which, of course, can be redirected). The report can be sent directly to a file by specifying a file name in the data record (record "b" in the example below).

The following code produces a listing suitable for printing ([from Section 5, Example 2](#)):

```
0 DEFINITION
1 define_record type=a
2 define_field field=9; length=25; default="Listing of Individuals\n"
2 define_field field=1; length=3; default="ID"
2 define_field field=2; length=20; default="Name"
2 define_field field=3; length=4; default="Sex"
2 define_field field=4; length=12; justify=R; default="Birth date"
2 define_field field=41; length=2
2 define_field field=5; length=15; default="Birth place"
1 define_record type=b ; lines=55; hdng=a
2 define_field field=1; length=3; default="No ID"
2 define_field field=2; length=20; default="Name Missing"
2 define_field field=3; length=4
2 define_field field=4; length=12; justify=r
2 define_field field=41; length=2
2 define_field field=5; length=15
0 HEAD
1 - init_record type=a
...
```

- C In the define\_file for record "b", the "hdng=a" ties the two files together for the combined output of headings and data.
- C To provide spacing between columns, the fields are larger than necessary, and blank fields were added (field 41).
- C The "\n" in the 3rd line causes a carriage return/line feed, to give two lines of headings.
- C The field numbers in the two records do not have to correspond. The order in which they are defined, and their lengths determine the data and heading alignment.
- C The "lines=55" parameter in the file definition for "b" specifies the number of lines on a page; that is, the number of lines between form feeds. In fact, the heading lines, and two

less than the specified number of data lines are printed. So the exact number of lines specified is printed only if there are exactly 2 heading lines.

- C The `init_record` for record type "a" is not really needed since all records are initialized automatically for the first time when they are defined.



## 4 Reference

### 4.1 GEDCOM to Fixed Format File Conversion Utility Command Syntax

The following is the syntax for executing the GEDCOM to Fixed Format File Conversion Utility

```
{ProgramName} {GEDCOM input file} {control file} {[output file|LPTn:]}
```

### 4.2 General Directive Syntax

The directive syntax is:

```
level# [+|-] directive parm1=value1; parm2=value2; . . .
```

- C Directives are written in lowercase so they are easily differentiated from tags (which are normally in uppercase).
- C The "+" and "-" are used for directives whose execution is triggered by the occurrence of an associated tag. The "-" indicates that the directive is to be executed immediately when an occurrence of the associated tag is found. The "+" indicates that the directive is to be executed after all of the subordinates of the associated tag have been processed. The "+" and "-" are not used for directives in the DEFINITION section.
- C Multiple parameters are separated by a semicolon.
- C The directive level number (level#) must be exactly one less than the level number of its associated tag. For example, for a directive associated with a birth date, we might have:

```
0 INDI
. . .
1 BIRT
2 DATE
3 - put_record type=a; field=3
```

- C The parameter keywords may be upper or lowercase, but the directive must be lowercase. For example, the following is correct:

```
3 + put_record TYPE=A; FIELD=3
```

- C All parameter values must account for leading and trailing spaces by enclosing the total field value inside double quotes, for example `PARSE=" | "` (space, vertical bar, space).

C Text following a /\* is treated as comments, that is, it is ignored. For example, in:

```
2 define_field field=2; length=0 /* Individual's Name
```

the "/\* Individual's Name" serves as a comment. (However, the GEDCOM syntax requires that each line start with a level number followed by something which does not violate the rules for a tag, so a comment cannot be on a line by itself.)

### 4.3 Syntax of Directives (in Alphabetic Order)

## **add\_to\_count**

### **Description**

The GEDCOM to Fixed Format File Conversion Utility supports up to 300 counters, 1 through 300. They can be used to count occurrences of tags (number of individuals, families, children, etc.) or number of output records (see [put\\_record](#)). The add\_to\_count directive increments a specified counter by a specified amount.

### **Parameter List**

The parameter list includes the following key words set to typical values:

counter=4	Counter number 1 through 300 to be incremented. A maximum of 300 counters can be defined.
add=1	The amount to increment the counter.
counter=RFN	This refers to a special global RFN counter that is converted to base 30 when retrieved for output by put_field.

### **Examples**

[See Section 3.6](#)

## **conc\_constant**

### **Description**

This directive concatenates a value to the end of a given field. To use the this directive, a field from the GEDCOM input file is first read into an output field using a prior put\_field directive.

### **Parameters**

type=a            Output record identifier, “a” through “e”. A maximum of five records can be defined.

field=3            The field number of the field that the data following the “value=” parameter will be concatenated to.

value=”-data”    A value to be concatenated to the data specified by the “field=” parameter.

### **Example**

2 - conc\_constant type=a; field=5; value=”good boy”

The above directive will concatenate ‘good boy’ to the end of the field value in field 5 of record type a.

## define\_field

### Description

The define\_field directive defines the characteristics of a field within the output record. The characteristics are length, justification, fill character to be used in fixed length fields, and default value assigned when the field is not present in the input GEDCOM record.

### Parameter List

The parameter list includes the following key words set to typical values:

field=1	ID number used to reference the field in other directives.
length=0	Field length. 0 indicates a variable length field.
justify=r	valid values are "r, L, u and p" r= will right justify the data for fixed length fields. L= will suppress leading blanks or zeroes and left justify. u= will convert to upper case. p= will convert to lower case except for characters which follow blanks, commas, periods, or are at the beginning of a line. These are converted to upper case.
fill=" "	Fill character used for fixed length fields.
default="Missing"	Specifies the value of the field, if no data is found in the GEDCOM input file for this field. When using the default values as headings for another record type, a special designator "\"n" (back slash n) causes a CRLF (carriage return + line feed) to be included in the text.

### Examples

[See Section 3.2](#)

### Notes

- C define\_field directives must be subordinate to a define\_record directive
- C The order of the fields in the output record is the order in which they were defined within the define record structure, not the field ID number order.

## **define\_record**

### **Description**

The `define_record` directive is used to define the characteristics of an output record. Subordinate `define_field` directives must be used to define the characteristics of each field in the record. All fields extracted from the GEDCOM input file are placed in a memory pool associated with the record identifier for that record. This pool space is only reclaimed when an `init_record` directive is used for that record identifier.

### **Parameter List**

The parameter list includes the following key words set to typical values:

<code>type=a</code>	Output record identifier, "a" through "e". A maximum of five records can be defined
<code>file=c:\dataout.txt</code>	Name of the file to which records will be written. A special case of <code>file=</code> parameter is either <code>file="output.EXT"</code> or <code>file="input.EXT"</code> . When either of these forms are used the output file name will be either the default input file name or the default output file name given on the command line, depending whether " <code>file=output</code> " or " <code>file=input</code> " was used, and the extension given by this file parameter. In this instance the extension is EXT.
<code>fdlim=", "</code>	Field delimiter, one or more characters (not a semicolon)

The following are optional parameters used for page control when creating a printed report from a GEDCOM file:

<code>Lines=55</code>	Number of lines on a page (actually a page will contain the heading lines(s), and this number minus 2 data lines).
<code>hdng=b</code>	Column headings are provided by the default field values given in the record definition whose record ID is specified here.

### **Examples**

[See Section 3.2](#)

### **Notes**

- C     The 0 DEFINITION record must be the first record in the control file, followed by the `define_record` directives

- C A `define_record` directive must be followed by one or more subordinate `define_field` directives
- C Field delimiters are applied only when the field is variable length, as indicated by `LENGTH=0`.
- C There must be a separate record definition to match any record type specified by an HDNG parameter.

## **if\_data\_put\_record**

### **Description**

To use the this directive, a field from the GEDCOM input file is first read into an output field using a prior put\_field directive. The if\_data\_put\_record directive checks to see if data in the record specified by the “ttype=” parameter is empty. If it is not empty and it is not equal to the default field value string provided by define\_field, then this directive will call the put\_record directive to write out a record specified by the “type=” parameter.

### **Parameter List**

The parameter list includes the following key word set to a typical value:

type=a	Record identifier of the record to be initialized.
--------	--

### **Example of usage**

- if\_data\_put\_record type=a; field=5; ttype=b

This example checks field 5 in record type b and if it contains data other than the default data then a record of type a is written through an automatic call to put\_record.



## init\_record

### Description

The init\_record directive initializes the specified output record by setting its fields to the default values and releasing all memory pool space previously used for this record type. The fields of a record can be overwritten with new values, and the record written out without initializing the record each time it is used. However, **when a field is "overwritten" the memory it used is not reused or released. Additional space is acquired. Repeated use of a record without initialization can cause the utility to run out of memory, and terminate.**

### Parameter List

The parameter list includes the following key word set to a typical value:

type=a	Record identifier of the record to be initialized.
--------	--

### Examples

[See Section 3.3](#)



## **look\_at\_field**

### **Description**

The look\_at\_field directive allows the operator to look at a specified field value and to either accept it or reject it. If it is rejected then the operator can modify the field value from the keyboard.

### **Parameter List**

The parameter list includes the following key word set to a typical value:

type=a	Record identifier of the record to be initialized.
field=3	

### **Examples**

- look\_at\_field type=a; field=3

This directive will print field 3 from record type a to the screen or console. The operator is given the option to accept it or reject it. If it is rejected the operator is then allowed to input the desired value.

## order\_field

### Description

To use the order\_field directive, a field from the GEDCOM input file is first read into an output field with a put\_field. An order\_field is then used to separate the field into sub-fields, based on either a one character delimiter specified by the "parse=" parameter or by a positional mask also specified by the "parse=" parameter. Each sub-field is placed in a specified output field (one of the sub-fields can be "moved" into the source field itself). If needed, the sub-fields may then be recombined in a different order and with any delimiter by using the "fetch" and "conc" parameters of the put\_field directive.

A variation of order\_field separates the field into sub-fields, reverses their order, and replaces them in the source field with the same delimiter (order=reverse). The order\_field directive is, also, able to translate an alphanumeric date (4 MAR 1905) into a numeric date (1905 03 04).  
(date=numeric)

### Parameter List

The parameter list includes the following key words set to typical values:

type=a	Record identifier of the record containing both the source and receiving fields.
field=1	ID of the field containing the source data to be parsed into sub-fields.
order=4,5,6	Field ID's of the receiving fields. As the sub-fields are found from left to right, they are placed in the receiving fields in the order listed.
order=reverse	Reverses the sub-fields, separates them with the same delimiter, and places the result back in the source field.
parse=","	Specifies the character that delimits the sub-fields.
parse="a bbb cccccc "	This is a positional mask used for parsing instead of the character delimiter. In this mask the first character would be parsed into subfield 1, the next 3 characters into subfield 2, and the next 6 characters into subfield 3. Note if the special character <angle brackets> surround one of the sets of characters being parsed, it will not be counted by positional mask, but placed as part of the subfield with the character it surrounded. For example, F<123>456789 would parse in to F, <123>, and 456789
date=numeric	Converts an alpha month to numeric.

fldcnt=2

This will cause the directive to stop parsing after reaching the specified number of subfields. Note, the “fldcnt=” parameter only works for token parsing, not positional.

### **Examples**

[See Section 3.5.](#)

### **Notes**

- C The order\_field directive applies to a field already stored in an output field by put\_field, thus the directive usually follows a put\_field.
- C The order\_field directive can handle a maximum of 9 sub-fields.

## put\_field

### Description

The put\_field directive reads an input GEDCOM tag value into to a specified output field. This directive also makes it possible to store data in an output field, and then copy it into another field in the same or a different record type ("fetch" parameter). For example, the cross reference ID for a family record could be stored in a "work area" record and then fetched and placed in separate records for each family member.

### Parameter List

The parameter list includes the following keywords set to typical values:

type=a	Record identifier containing the desired output field.
field=1	Output field ID number.
conc=", "	Causes the data from the GEDCOM input to be concatenated to the end of the data that already resides in the specified field. The data between the quotes is used as the delimiter between the existing data and the new data being concatenated. This delimiter may be multiple characters, such as, a comma and space for separating place names.
default="constant field value"	Specifies a value which is assigned to this field if the input data for this context if the length of the context is zero.
split="14,26"	This has the format split="length,new-field". This causes the data from the GEDCOM input to be split into two different fields if it is longer than the length parameter. If it is it puts the remaining characters into the field specified by the new-field parameter. If the new-field parameter is 0 then the data is chopped at the size of the length parameter.

The following two parameters are used to retrieve a field from an output record and place it in the record and field specified in the type and field parameters above.

ttype=b	The output record from which a field is to be retrieved
fetch=1	The field to be retrieved.

The following two parameters are used to retrieve a count and place it in the record and field specified in the type and field parameters above.

fetch=count	Specifies a count is to be retrieved.
-------------	---------------------------------------

counter=3

The counter number from which the count is to be retrieved.

### **Examples**

Sections [3.4](#), [3.5](#), and [3.6](#)

### **Note**

- C Using a simple put\_field saves the GEDCOM line value. Use the put\_xref directive to get to any, or all, of the cross reference (xref), tag (tag), or pointer (ptr). That directive strips enclosing @ signs from both the cross reference and the pointer.

## put\_multi\_field

### Description

The put\_multi\_field directive is required when the same tag occurs multiple times without intervening tags. This occurs often with the note structure. Multi field puts use several output fields and a group pointer that keeps track of the next output field to be used. The group pointer is reset ([reset\\_multi\\_fld\\_grp](#)) before each group of multiple occurring tags is processed, and is incremented for each multi field put. Since more than one set of multiple occurring tags can exist within a structure, there are five multi field group identifiers available (1 through 5).

**Warning:** When the specified fields are all filled, any values from further occurrences of the tag are ignored. If the reset\_multi\_fld\_grp directive is not used to reset the group pointer at the appropriate time, then only the first set of multiple fields will be saved.

### Parameter List

The parameter list includes the following key words set to typical values:

type=a	Output record identifier.
field=1,2,3,4	The set of output fields being used.
group=1	Identifier of the group pointer being used (must be from 1 to 5).

### Examples

[See Section 3.4.3](#)



## **put\_record**

### **Description**

This directive writes an output record. The order of the fields in the output record is the order in which they were defined within the define record structure, not the field number order.

### **Parameter List**

The parameter list includes the following key words set to typical values:

type=a            Record identifier.

The following parameters can be used to count when a record is written. If selected records are being written (see [select\\_record](#)), the counter is incremented only for records selected (actually written):

counter=2            The counter to be incremented.  
add=1                Amount by which the counter is incremented.

### **Examples**

[See Section 3.3](#)

### **Note**

## **put\_xref**

### **Description**

This directive is like the put\_field directive in that it writes data from the GEDCOM input file to specified fields within the specified output record. However, it writes cross reference ID's, pointers, and tags to the output record.

### **Parameter List**

The parameter list includes the following key words set to their default value:

type=a	Output record identifier.
xref=1	Writes the cross reference ID to the field number. Enclosing @ signs are stripped.
tag=2	Writes the GEDCOM tag to the field number.
ptr=3	Writes the pointer value to the field number. Enclosing @ signs are stripped.

Any parameter assigned to field 0, or which is omitted, is not written to the output record.

### **Example**

[See Section 3.4.2](#)

### **Note**

- C     A put\_xref can put data into multiple fields, up to 3, by specifying values for xref, tag, and/or ptr in a single xref\_put directive.

## **reset\_count**

### **Description**

Reset\_count set the specified counter to the initial value indicated.

### **Parameter List**

The parameter list includes the following key words set to typical values:

counter=3	Counter to be initialized, 1 through 300.
reset=1	The value to which the counter is set.

### **Examples**

[See Section 3.6](#)

## **reset\_multi\_fld\_grp**

### **Description**

This directive resets the current field pointer for the specified multi field group. (See [put\\_multi\\_field](#)).

### **Parameter List**

The parameter list includes the following key word set to a typical value:

group=1	Resets the field pointer for multiple tag groups back to the beginning field.
---------	---

### **Examples**

See Section 3.4.3

### **Note**

C      There can be up to 5 multi field groups active at a time.

## **select\_record**

### **Description**

The select\_record directive provides for writing only selected records to an output file. Each record type is controlled separately.

In the start/stop mode of record selection, a start parameter turns a write switch on, and all records are written until a stop parameter turns the switch off.

In the include/exclude mode, each record is tested individually to see if it satisfies the conditions for selection. A write switch, separate from start/stop switch, is used for this.

### **Parameter List**

The parameter list includes the following key words set to typical values:

type=a            The record type for which records are being selected.

If a counter is to be used for comparison, use:

counter=5            The counter to be compared.

If a field is to be used for comparison, use:

ttype=e            The record containing the field to test

fetch=2            The field to test

For start/stop selection, use:

start=10            Value used to turn the start/stop write switch on. (If a text field is being used to, enclose the value in quotes, i.e. start="john /smith/".)

stop=25            Value used to turn the start/stop write switch off.

flush=25           Same as "stop", except it terminates the processing of the input file.

For include/exclude selection, use:

include="smith"      Value which causes a record to be written.

exclude="jones"      Value which causes a record to not be written.

To specify how the value and field (or counter) are to be compared, use:

when=eq	Operator to use in comparing. The acceptable values are:	
	eq - equal to	ne - not equal to
	lt - less than	le - less than or equal to
	gt - greater than	ge - greater than or equal to

## **Examples**

[See Section 3.7](#)

## **Notes**

- C Start/stop selection and include/exclude selection are controlled by independent switches, and can be used together on the same record type. A record is printed if both are on.
- C The include/exclude mode of selection is controlled by a write switch that can be set by any number of successive "select\_record include/exclude" directives. The order of multiple include/exclude selection directives can be significant. The first include/exclude sets the switch for both explicit and implied conditions. For example,

```
select_record include="smith"; when=eq; . . .
```

will set the switch on for smith, and off for any other value. Subsequent include/excludes only honor the explicit condition, that is, include="jones" will set the switch on for jones, but will not set it off for other values (smith); thus it does not override the selection of "smith" and both smiths and jones' will be written. In the following:

```
select_record include="g"; when=ge; . . .  
select_record exclude="n"; when=ge; . . .
```

the first statement turns the write switch on for all fields starting with "g" through "z", and off for all others. The second turns the switch off for all fields starting with "n" through "z" (but does not effect the switch for others). Thus the switch will be on for field values starting with "g" through "m". If the order of these statements is reversed, the result is quite different; the switch will be on for all field values.

## 4.4 Error Codes

### Memory Errors:

<u>Error #</u>	<u>Error definition</u>
1	Memory error, unusual error, may be bad grammar file or bad GEDCOM data file.
2	Memory error, not enough memory to create space pool.
3	Memory error, ran out of space in pool, could not get any more.
304	GEDCOM record too long for available memory.

### Input/Output Errors:

<u>Error #</u>	<u>Error definition</u>
100-199	Any errors in this range are due to input/output errors reading or writing a file.

### GEDCOM Format Errors:

<u>Error #</u>	<u>Error definition</u>
201	Invalid character found before the level number.
202	The cross reference value exceeded maximum length.
203	Tag contained invalid characters. The characters in tags are limited to ascii characters 32 through 127.
204	Tag length exceeded maximum length.
205	Found an opening @ in the cross reference but found a space before finding the closing @.
216	Invalid character was found before the level number.
2060	End of the buffer was found while looking for the level number.
303	Found a directive syntax in the grammar that is not recognized by the program.

## 5 Complete Examples

### GEDCOM Input

The following is sample GEDCOM input for the examples which follow. After the control code for each example, the output produced from this input is shown.

```
0 HEAD
1 SOUR EFT
2 VERS 1.1
2 NAME Whatever
1 DEST PAF
1 DATE 15 Nov 1995
1 FILE EXAMPLE.GED
1 GEDC
2 VERS 4.0
2 FORM LINEAGE-LINKED
0 @I1@ INDI
1 NAME Jane /Austin/
1 SEX F
1 BIRT
2 DATE 17 DEC 1934
2 PLAC Long Beach,Los Angeles,CA
1 FAMS @F1@
0 @I2@ INDI
1 NAME Oliver /TWIST/
1 SEX M
1 BIRT
2 DATE 17 MAR 1935
2 PLAC Ogden,Weber,Utah
1 BAPL
2 DATE 2 MAY 1943
1 FAMS @F1@
1 FAMC @F782@
0 @I3@ INDI
1 NAME Sheila /TWIST/
1 SEX F
1 BIRT
2 DATE 25 SEP 1956
2 PLAC Whittier,Los Angeles,Calif
1 FAMC @F1@
0 @I5@ INDI
1 NAME John Lester /TWIST/
1 SEX M
1 BIRT
2 DATE 4 MAY 1960
2 PLAC Provo,Utah,UT
1 FAMS @F783@
1 FAMC @F1@
0 @F1@ FAM
1 NOTE This is a note
2 CONT with continuations:
2 CONT Note segment 3
2 CONT Note segment 4
2 CONT Note segment 5
1 HUSB @I2@
1 WIFE @I1@
1 CHIL @I3@
1 CHIL @I4@
1 CHIL @I5@
1 CHIL @I6@
1 CHIL @I7@
1 CHIL @I2059@
1 MARR
2 DATE 3 JUN 1955
2 PLAC Las Vegas,Clark,Nevada
0 @F25@ FAM
1 NOTE Sample note
1 HUSB @I64@
1 WIFE @I65@
1 MARR
0 @F783@ FAM
1 HUSB @I5@
1 WIFE @I2060@
1 CHIL @I2061@
1 CHIL @I2062@
1 CHIL @I2063@
1 MARR
2 DATE 30 OCT 1981
2 PLAC Provo,Utah,UT
0 TRLR
```



These examples are indented and commented to, hopefully, make them more readable and understandable. Most of the code samples in Section 3 can be found in context in these examples.

### Example 1:

#### Control Code:

#### 0 DEFINITION

```
1 define_record file=c:\g2f\people.txt; type=a          /* Individuals file
  2 define_field field=1; length=3; default="No ID"      /* Individual's ID
  2 define_field field=2; length=20; default="Name Missing" /* Full name
  2 define_field field=3; length=2                      /* Sex
  2 define_field field=4; length=12; justify=r          /* Birth date: 4 Mar 1945
  2 define_field field=41; length=12; justify=r         /* Birth date: 1945 03 04
  2 define_field field=42; length=2                    /* Spacing for readability of output
  2 define_field field=5; length=20                    /* Birth place
  2 define_field field=6; length=6                     /* Family ID in which this individual is a spouse
  2 define_field field=7; length=6                     /* Family ID in which this individual is a child
1 define_record type=b; file=c:\g2f\families.txt; fdlim="|" /* Families file
  2 define_field field=1; length=0; default="Missing ID" /* Family ID
  2 define_field field=2; length=0                     /* Father's ID
  2 define_field field=3; length=0                     /* Mother's ID
  2 define_field field=4; length=0                     /* Marriage date
  2 define_field field=5; length=0                     /* Marriage place: City, County, State, Country
  2 define_field field=51; length=0                    /* Marriage place: Country, State, County, City
  2 define_field field=6; length=0                     /* Family Notes
  2 define_field field=7; length=0                     /* Family Notes
  2 define_field field=8; length=0                     /* Family Notes
  2 define_field field=9; length=0                     /* Family Notes
1 define_record type=c; file=c:\g2f\children.txt; fdlim="|" /* Children file
  2 define_field field=1; length=0                     /* Family ID
  2 define_field field=2; length=0                     /* Child ID
1 define_record type=e                                /* Work area temporary storage
  2 define_field field=1; length=0
  2 define_field field=2; length=0
  2 define_field field=3; length=0
```

#### 0 INDI

```
1 - init_record type=a                                /* Clear record "a" and release space
1 - put_xref type=a; xref=1                            /* Individual's ID into individual record
1 - init_record type=e                                /* Clear temporary storage area and release space
1 + put_record type=a                                  /* Write record after all subordinate tags processed
1 NAME
  2 - put_field type=e; field=1                        /* Name into temp. storage
```

```

2 - order\_field type=e; field=1; order=2,3; parse="/"      /* Break name apart
2 - put_field type=a; field=2; ttype=e; fetch=3           /* Surname into field 2
2 - put_field type=a; field=2; conc=", "; ttype=e; fetch=2 /* Given names into field 2
1 SEX
2 - put_field type=a; field=3                            /* Sex into individual's record
1 BIRT
2 DATE
3 - put_field type=a; field=4                            /* Birth date into individual's record
3 - put_field type=a; field=41                           /* Put another Birth date to reverse for sorting
3 - order_field type=a; field=41; order=reverse; date=numeric; parse=" "
2 PLAC
3 - put_field type=a; field=5                            /* Birth place into individual record
1 FAMS
2 - put_xref type=a; ptr=6                               /* Family ID where this individual is a spouse
1 FAMC
2 - put_xref type=a; ptr=7                               /* Family ID where this individual is a child
0 FAM
1 - init_record type=b                                  /* Clear record "b" and release space
1 - put_xref type=b; xref=1                              /* Family ID into family record
1 - reset\_multi\_fld\_grp group=1                        /* Reset field pointer for multiple notes fields
1 + put_record type=b                                   /* Write record after all subordinate tags processed
1 NOTE
2 - put\_multi\_field type=b; group=1; field=6,7,8,9      /* Note into field 6
2 CONT
3 - put_multi_field type=b; group=1; field=6,7,8,9 /* Multiple continuations fields
1 HUSB
2 - put_xref type=b; ptr=2                               /* Father pointer into family record (field 2)
1 WIFE
2 - put_xref type=b; ptr=3                               /* Mother pointer into family record (field 3)
1 CHIL
2 - init_record type=c                                  /* Clear record "c" and release space
2 - put_field type=c; field=1; ttype=b; fetch=1          /* Get Family ID from record "b"
2 - put_xref type=c; ptr=2                               /* Put child pointer into children file
2 - put_record type=c                                   /* Write record "c"
1 MARR
2 DATE
3 - put_field type=b; field=4                            /* Marriage date into families file
2 PLAC
3 - put_field type=b; field=5                            /* Marriage place into families file
3 - put_field type=b; field=51                           /* Another marriage place to reverse for sorting
3 - order_field type=b; field=51; parse=","; order=reverse

```

## Output:

### people.txt

I1	Austin, Jane	F	17	DEC	1934	1934	12	17	Long Beach, Los Angeles	F1	
I2	TWIST, Oliver	M	17	MAR	1935	1935	03	17	Ogden, Weber, Utah	F1	F782
I3	TWIST, Sheila	F	25	SEP	1956	1956	09	25	Whittier, Los Angeles	F1	
I5	TWIST, John Lester	M	4	MAY	1960	1960	05	04	Provo, Utah, UT	F783	F1

### families.txt

F1|I2|I1|3 JUN 1955|Las Vegas, Clark, Nevada|Nevada, Clark, Las Vegas|This is a note|with continuations:  
|Note segment 3|Note segment 4|  
F25|I64|I65||||Sample note||||  
F783|I5|I2060|30 OCT 1981|Provo, Utah, UT|UT, Utah, Provo||||

### children.txt

F1|I3|  
F1|I4|  
F1|I5|  
F1|I6|  
F1|I7|  
F1|I2059|  
F783|I2061|  
F783|I2062|  
F783|I2063|

## Example 2:

### Control code:

#### 0 DEFINITION

```
1 define_record type=a /* Headings for individuals listing
2 define_field field=9; length=25; default="Listing of Individuals\n"
2 define_field field=1; length=3; default="ID"
2 define_field field=2; length=20; default="Name"
2 define_field field=3; length=4; default="Sex"
2 define_field field=4; length=12; justify=R; default="Birth date"
2 define_field field=41; length=2
2 define_field field=42; length=13; default="Birth(sort)"
2 define_field field=5; length=15; default="Birth place"
2 define_field field=51; length=2
2 define_field field=6; length=7; default="Child"
2 define_field field=7; length=6; default="Spouse"
1 define_record type=b ; lines=55; hdng=a /* Data record for individuals listing
2 define_field field=1; length=3; default="No ID" /* Individual's ID
2 define_field field=2; length=20; default="Name Missing" /* Name
2 define_field field=3; length=4 /* Sex
2 define_field field=4; length=12; justify=r /* Birth date: 4 Mar 1945
2 define_field field=41; length=2 /* Spacing
2 define_field field=42; length=13 /* Birth date: 1945 03 04
2 define_field field=5; length=15 /* Birth Place
2 define_field field=51; length=2 /* Spacing
2 define_field field=6; length=7 /* Family ID where this individual is a Child
2 define_field field=7; length=6 /* Family ID where this individual is a Spouse
1 define_record type=c; file=c:\g2f\families.txt; fdlim=""," /* Families file; notice delimiter
2 define_field field=91; length=1; default=""" /* Leading "
2 define_field field=1; length=0; default="Missing ID"
2 define_field field=2; length=0 /* This is an example of a comma delimited
2 define_field field=3; length=0 /* file with the fields in quotes, that is:
2 define_field field=4; length=0 /* "field1","field2","field3", etc.
2 define_field field=5; length=0
2 define_field field=6; length=0 /* Family notes concatenated in one field
2 define_field field=92; length=1; default=""" /* Trailing "
1 define_record file=c:\g2f\fammembr.txt; type=d /* Family members file
2 define_field field=1; length=5 /* Family ID
2 define_field field=2; length=6 /* Relationship tag, HUSB, WIFE, or CHIL
2 define_field field=3; length=5 /* Individual ID
1 define_record type=e /* Temporary storage area
2 define_field field=1; length=0
2 define_field field=2; length=0
2 define_field field=3; length=0
```

#### 0 HEAD

```

1 - init_record type=a                                /* Clear & place headings (defaults) in heading rec.
0 INDI
1 - init_record type=b                                /* Clear individuals record and release space
1 - init_record type=e                                /* Clear temp. storage and release space
1 - put_xref type=b; xref=1                            /* Individual's ID into individual's records
1 + put_record type=b                                /* Write record after subord. tags processed
1 NAME
2 - put_field type=e; field=1                          /* Read name into temp. storage
2 - order_field type=e; field=1; order=2,3; parse="/"    /* Break name up
2 - put_field type=b; field=2; ttype=e; fetch=3          /* Surname into field 2
2 - put_field type=b; field=2; conc=" "; ttype=e; fetch=2 /* Given names into field 2
1 SEX
2 - put_field type=b; field=3                          /* Sex into individual's record
1 BIRT
2 DATE
3 - put_field type=b; field=4                          /* Birth date into individual's record
3 - put_field type=b; field=42                         /* Get Birth date, then reverse it
3 - order_field type=b; field=42; order=reverse; date=numeric; parse=" "
2 PLAC
3 - put_field type=b; field=5                          /* Birth place into individual's record
1 FAMC
2 - put_xref type=b; ptr=6                            /* Pointer to family where this individual is a child
1 FAMS
2 - put_xref type=b; ptr=7                            /* Pointer to family where this individual is a spouse
0 FAM
1 - init_record type=c                                /* Clear record family record and free space
1 - put_xref type=c; xref=1                            /* Family ID into family record
1 + put_record type=c                                /* Write record after all subordinate tags are processed
1 - init_record type=d                                /* Clear record family member record and free space
1 - put_xref type=d; xref=1                            /* Family ID into family member record
1 NOTE
2 - put_field type=c; field=6                          /* Get first part of note into field 6
2 CONT
3 - put_field type=c; field=6; conc=" "                /* Concatenate continuations into field 6
1 HUSB
2 - put_xref type=c; ptr=2                            /* Father's ID into family record
2 - put_xref type=d; tag=2; ptr=3                      /* Father's ID & HUSB into family member record
2 - put_record type=d                                /* Write out family member record
1 WIFE
2 - put_xref type=c; ptr=3                            /* Mother's ID into record family record
2 - put_xref type=d; tag=2; ptr=3                      /* Mother's ID & WIFE into family member record
2 - put_record type=d                                /* Write out family member record
1 MARR
2 DATE
3 - put_field type=c; field=4                          /* Marriage date into family record
2 PLAC

```

```

3 - put_field type=c; field=5          /* Marriage place into family record
1 CHIL
2 - put_xref type=d; tag=2; ptr=3      /* Child's ID & CHIL into family member record
2 - put_record type=d                  /* Write out family member record

```

### Output:

#### Printed report:

##### Listing of Individuals

ID	Name	Sex	Birth date	Birth(sort)	Birth place	Child	Spouse
I1	Austin, Jane	F	17 DEC 1934	1934 12 17	Long Beach, Los		F1
I2	TWIST, Oliver	M	17 MAR 1935	1935 03 17	Ogden, Weber, Uta	F782	F1
I3	TWIST, Sheila	F	25 SEP 1956	1956 09 25	Whittier, Los An	F1	
I5	TWIST, John Lester	M	4 MAY 1960	1960 05 04	Provo, Utah, UT	F1	F783

##### families.txt

```

"F1", "I2", "I1", "3 JUN 1955", "Las Vegas, Clark, Nevada",
  "This is a note with continuations: Note segment 3 Note segment 4 Note segment 5", ""
"F25", "I64", "I65", "", "", "Sample note", ""
"F783", "I5", "I2060", "30 OCT 1981", "Provo, Utah, UT", "", ""

```

##### fammembr.txt

```

F1    HUSB  I2
F1    WIFE  I1
F1    CHIL  I3
F1    CHIL  I4
F1    CHIL  I5
F1    CHIL  I6
F1    CHIL  I7
F1    CHIL  I2059
F25   HUSB  I64
F25   WIFE  I65
F783  HUSB  I5
F783  WIFE  I2060
F783  CHIL  I2061
F783  CHIL  I2062
F783  CHIL  I2063

```

### Example 3 Control Code:

#### 0 DEFINITION

```
1 define_record type=a /* Headings for summary record
2 define_field field=91; length=30; default="File Summary Information\n"
2 define_field field=1; length=8; default="Source"
2 define_field field=2; length=13; default="Destination"
2 define_field field=3; length=13; default="Date"
2 define_field field=4; length=13; default="File name"
2 define_field field=5; length=11; default="Individuals"; justify=r
2 define_field field=6; length=10; default="Families"; justify=r
1 define_record type=b; file=c:\g2f\summary.txt; hdng=a; lines=55 /* Summary record
2 define_field field=1; length=8 /* Input data source
2 define_field field=2; length=13 /* Intended destination
2 define_field field=3; length=13 /* Date file was produced
2 define_field field=4; length=13 /* Input file name
2 define_field field=5; length=11; justify=r /* Number of INDI records written out
2 define_field field=6; length=10; justify=r /* Number of FAM records read in
1 define_record type=c; file=c:\g2f\people.txt; fdlim="~" /* Individuals file
2 define_field field=1; length=0 /* Individual's ID
2 define_field field=2; length=0 /* Individual's name
1 define_record type=d; file=c:\g2f\families.txt; fdlim="~" /* Families file
2 define_field field=1; length=0 /* Sequential record number
2 define_field field=2; length=0 /* Family ID
2 define_field field=3; length=0 /* Father's ID
2 define_field field=4; length=0 /* Mother's ID
2 define_field field=5; length=0 /* Number of children
1 define_record type=e /* Work area
2 define_field field=1; length=0
2 define_field field=2; length=0
```

#### 0 HEAD

```
1 - init_record type=a /* Initialize headings for summary listing
1 - init_record type=b /* Initialize summary record
1 - reset_count counter=1; reset=0 /* Initialize counter, number of INDI records written
1 - reset_count counter=2; reset=0 /* Initialize counter, number of FAM records input
```

#### 1 SOUR

```
2 - put_field type=b; field=1 /* Input source into summary record
```

#### 1 DEST

```
2 - put_field type=b; field=2 /* Intended destination into summary record
```

#### 1 DATE

```
2 - put_field type=b; field=3 /* Creation date into summary record
```

#### 1 FILE

```
2 - put_field type=b; field=4 /* Input file name into summary record
```

#### 0 INDI

```
1 - init_record type=c /* Initialize individual's record
```

```

1 - init_record type=e /* Initialize work area
1 - put_xref type=c; xref=1 /* Individual's ID into individual's record
1 + select_record type=c; ttype=e; fetch=2; include="twist"; when=eq /* Test surname
1 + put_record type=c; counter=1; add=1 /* Write only Twists; count records written
1 NAME
  2 - put_field type=c; field=2 /* Individual's full name into individual's record
  2 - put_field type=e; field=1 /* Full name in the work area
  2 - order_field type=e; field=1; order=1,2; parse="/" /* Split name; surname in field 2
0 FAM
  1 - init_record type=d /* Initialize family record
  1 - add_to_count counter=2; add=1 /* Count FAM input records
  1 - put_field type=d; field=1; fetch=count; counter=2 /* sequence number the family records
  1 - put_xref xref=2; type=d /* Family ID into family record
  1 - reset_count counter=3; reset=0 /* Initialize counter for children
  1 + put_field type=d; field=5; fetch=count; counter=3 /* Put count of children in family rec.
  1 + put_record type=d /* Write family record.
1 HUSB
  2 - put_xref type=d; ptr=3 /* Father ID into family record
1 WIFE
  2 - put_xref type=d; ptr=4 /* Mother ID into family record
1 CHIL
  2 - add_to_count counter=3; add=1 /* Count child
0 TRLR
  1 - put_field field=5; type=b; fetch=count; counter=1 /* # of individ output records in summary
  1 - put_field field=6; type=b; fetch=count; counter=2 /* # of FAM input records into summary
  1 - put_record type=b /* Write summary record

```

## Output

summary.txt

```

File Summary Information
Source  Destination  Date           File name      Individuals  Families
EFT     PAF              15 Nov 1995    EXAMPLE.GED    3            3

```

people.txt

```

I2~Oliver /TWIST/~
I3~Sheila /TWIST/~
I5~John Lester /TWIST/~

```

families.txt

```

1~F1~I2~I1~6~
2~F25~I64~I65~0~
3~F783~I5~I2060~3~

```



#### Example 4 Control code:

##### 0 DEFINITION

```
1 define_record type=a; file=c:\g2f\people.txt; fdlim="|" /* Individuals file
  2 define_field field=1; length=0 /* Individual output record sequence number
  2 define_field field=2; length=0 /* Individual's ID
  2 define_field field=3; length=0 /* Individual's name
1 define_record type=b; file=c:\g2f\families.txt; fdlim="|" /*Families file
  2 define_field field=1; length=0 /* FAM input record sequence number
  2 define_field field=2; length=0 /* Family ID
  2 define_field field=3; length=0 /* Father's ID
  2 define_field field=4; length=0 /* Mother's ID
1 define_record type=e /* Work area
  2 define_field field=1; length=0
  2 define_field field=2; length=0
```

##### 0 HEAD

```
1 - reset_count counter=1; reset=1 /* Initialize indiv. output record sequence number
1 - reset_count counter=2; reset=0 /* Initialize FAM input record counter
```

##### 0 INDI

```
1 - init_record type=a /* Initialize individual's output record
1 - init_record type=e /* Initialize work area
1 - put_xref type=a; xref=2 /* Individual's ID into individual's output record
1 + put_field type=a; field=1; fetch=count; counter=1 /* Put output rec. sequence # in output
1 + select_record type=a; counter=1; stop=21; when=eq /* Stop output after 20th record written
1 + select_record type=a; ttype=e; fetch=2; include="g"; when=ge /* Write g's thru z's
1 + select_record type=a; ttype=e; fetch=2; exclude="u"; when=ge /* Don't write u's thru z's
1 + put_record type=a; counter=1; add=1 /* Write g's thru t's; count output record
```

##### 1 NAME

```
2 - put_field type=a; field=3 /* Full name into individual's record
2 - put_field type=e; field=1 /* Put full name in work area
2 - order_field type=e; field=1; order=1,2; parse="/" /* Break up name; surname in field 2
```

##### 0 FAM

```
1 - init_record type=b /* Initialize family output record
1 - add_to_count counter=2; add=1 /* Count FAM input record
1 - put_field type=b; field=1; fetch=count; counter=2 /* Input record # in family output record
1 - put_xref type=b; xref=2 /* Family ID into family record
1 + select_record type=b; counter=2; start=2; when=eq /* Start writing output on 2nd record
1 + select_record type=b; counter=2; stop=4; when=eq /* Stop writing before 4th record
1 + put_record type=b /* Write family record
```

##### 1 HUSB

```
2 - put_xref type=b; ptr=3 /* Father's ID in family record
```

##### 1 WIFE

```
2 - put_xref type=b; ptr=4 /* Mother's ID in family record
```

Output:

people.txt

```
1|I2|Oliver /TWIST/|
2|I3|Sheila /TWIST/|
3|I5|John Lester /TWIST/|
```

families.txt

```
2|F25|I64|I65|
3|F783|I5|I2060|
```