PPMSORT: Parallel Open Systems Sort Fixed Format Record Based File Sorting User Guide and Reference Version 1

CML00003-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

December 15, 2020

Contents

1 Introduction

PPMSORT is a sort utility which can be configured to use the resources of an available machine in order to improve the sort times and throughput. PPMSORT sorts files which are expected to be made up of records with fixed or variable length records, but which adhere to a fixed layout such as is typical of files described and used in legacy systems such as PL/I and COBOL.

PPMSORT uses a configuration file in which the bounds on the resources that an instance of PPMSORT may use are stated.

From an application point of view, the format of the key specification follows the same syntax and semantics as is used on traditional host systems such as MVS. In addition it is possible to use the application meta-data to specify fields and hence to describe the sort requirements in a symbolic manner.

2 Configuration File

When PPMSORT is started, a suitable configuration file is sought which is used to supply the resource bound values; to specify the location of PPMSORT components; and to specify global parameters that are required to be applicable to all PPMSORT instances that use that configuration file. It is an error for no applicable configuration file to be found.

2.1 Configuration File Location

If the environment variable PPMSORTCONF is set, then it is expected to contain the path and file name of the configuration file to use. If this method of naming and locating the configuration file is used it takes precedence over all other methods of locating a usable configuration file. For example, in the following, the user is requesting that his own test version of the configuration file be used to govern the following sort:

```
[stephen@nomad sorttest]$ cp ppmsort.cfg testsort.cfg
[stephen@nomad sorttest]$ vi testsort.cfg
[stephen@nomad sorttest]$ export PPMSORTCONF=testsort.cfg
[stephen@nomad sorttest]$ ppmsort --input-file-name=/tmp/sortdata.bin \
        --output-file-name=/tmp/testdata.bin  \
        --record-length=$RECLEN  \
        --sort-key="SORT FIELDS=(11,23,CST,A)"
[ppmsort] $Id: ppmsort.c,v 1.23 2010/04/07 18:50:42 hayward Exp $
Copyright (c) 2007 by Code Magus Limited. All rights reserved.
mailto:stephen@codemagus.com, http://www.codemagus.com.
Total bytes input to sort process      = 1150000.
Total records input to sort process    = 50000.
```

```
Total bytes output from sort process   = 1150000.
Total records output from sort process = 50000.
[stephen@nomad sorttest]$
```

If the PPMSORTCONF environment variable is not set then the current working directory is checked for the presence of a file called ppmsort.cfg and should this file be present then it is processed as though it is a PPMSORT configuration file.

If, after checking the current working directory, a configuration file is not found, then the presence of the system wide PPMSORT configuration file /etc/ppmsort.cfg is determined and if present is used as the configuration file.

The reason for the various configuration files is to allow different sort situations to be used if required, but then to default to a common system wide or application wide configuration file for all requests for which a system or application wide configuration is applicable.

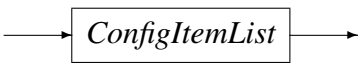
2.2 Configuration File Settings

Independent of the location of a PPMSORT configuration file, the contents always have the same format. A configuration file is a text file which adheres to a particular grammar. Comments and blank lines are allowed anywhere in a configuration file, except that comments may not appear between the letters of a keyword or string. A comment may appear anywhere on a line and the presence of a comment is introduced by two consecutive dashes (−−). A comment starts from the −− characters and continues up to and including the end of the line on which the −− appear. Comments cannot be continued and so should a comment need to span more than one line, subsequent lines should be prefixed with the −− characters.

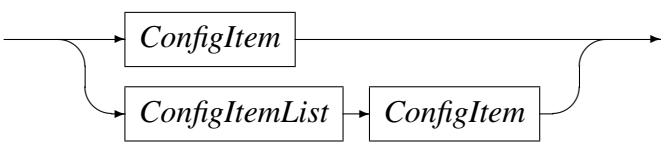
All configuration file options and values can be overridden as a command line argument to PPMSORT (See Section 3).

The following describes the syntax and corresponding semantics of PPMSORT configuration files.

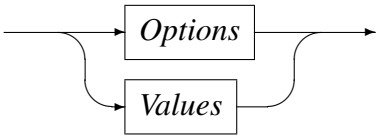
ConfigFile



ConfigItemList

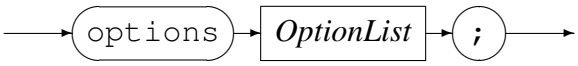


ConfigItem

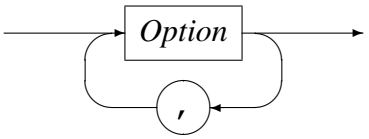


A configuration file provides values and options that will be used by a PPMSORT process instance. The file comprises a list of statements which are either option or flag statements or provide values to items. Options correspond to flags where a flag is set by the presence of an option keyword. Values are assigned by the configuration file by using the name of the value followed by an = sign followed by the item's value. Both option and value configuration file statements are terminated by a semi-colon.

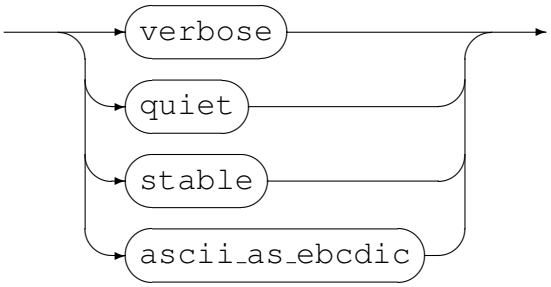
Options



OptionList



Option



An options statement is introduced using the options keyword. The elective options are indicated by a comma-separated list of keywords that correspond to the required option flags. There can be more than one options statement in a configuration file.

The verbose option indicates that all processing should proceed in a manner which maximises the diagnostic output produced. In particular all the internal process arguments are formatted and the intermediate sort-work files are not removed once they are no longer required for processing. The purpose of the verbose flag is to provide diagnostic output should problems be encountered either with the environment that PPMSORT operates in or with PPMSORT itself.

The quiet option is provided to have an alternative to the verbose option. The option has no effect and is supported as a comment.

The `stable` option causes the sort processing to operate in a stable manner, that is that records of equal key values retain their relative position in the output file.

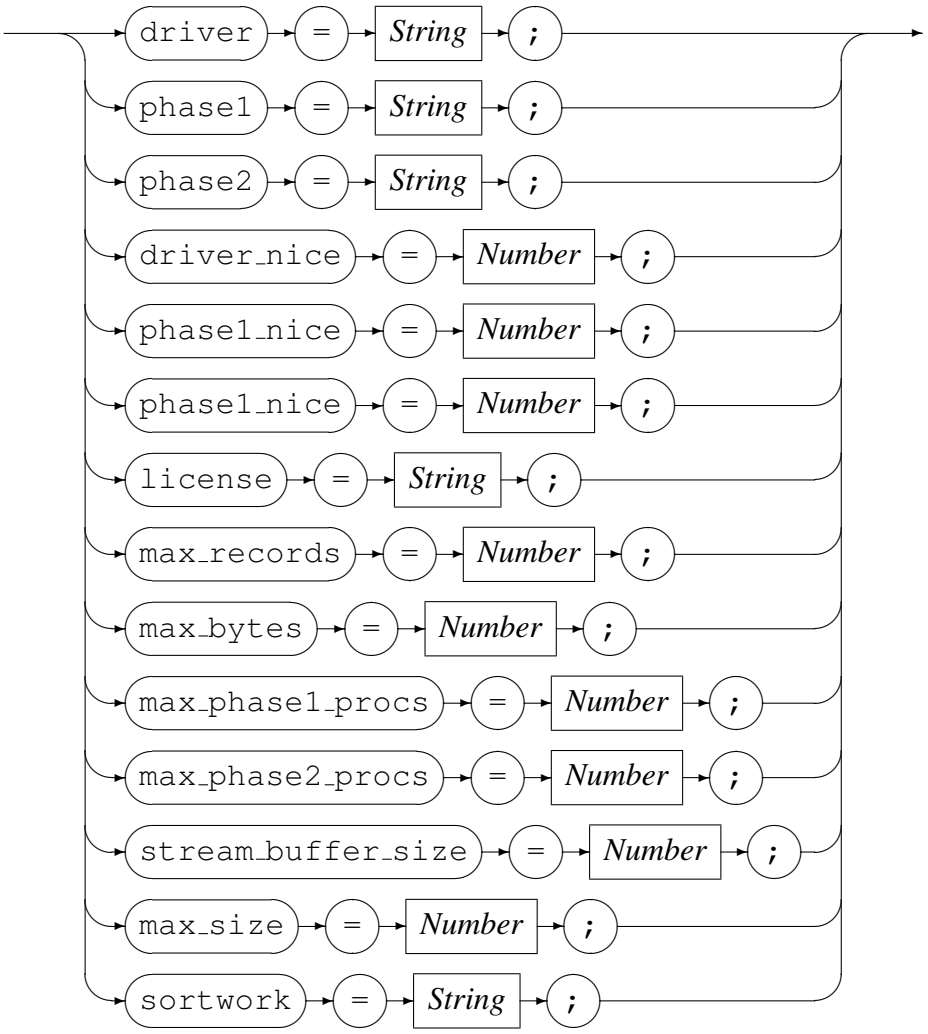
The `ascii_as_ebcdic` option treats all ASCII character data as the corresponding EBCDIC character data and applies the EBCDIC ordering of the graphic characters to the ASCII data. It is assumed that the character based data is ISO-8 data with graphic mappings according to IBM code page 819, and that the corresponding EBCDIC data graphic mappings are according to IBM code page 1047.

The primary affect that this option will have on the collating sequence of character data is that the digits will fall after all alphabetic characters and that the lower case characters will appear before all the upper case characters. This is different to the ASCII or ISO-8 assignments in which the digits precede the upper case characters and which in turn precede the lower case characters. Additionally, the relative placement of the other characters such as the control and special graphic characters are ordered according to EBCDIC IBM code page 1047 code points rather than the ISO-8 IBM code page 819 code points.

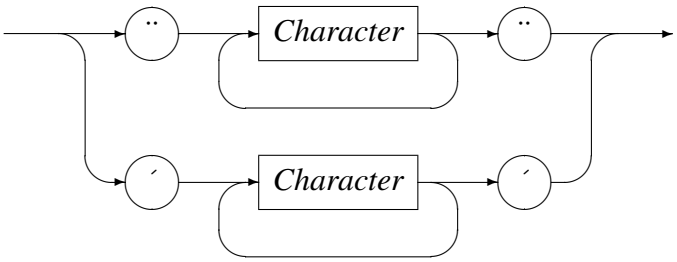
As an example, the following `options` statement requests that `PPMSORT` behave as a stable sort and that the characters' collating sequence is to conform the IBM code page 1047, assuming that the original data comprised of IBM code page 819 graphics.

```
options stable, ascii_as_ebcdic;
```

Values

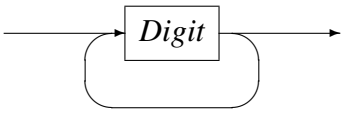


String



In the first instance a *Character* is any single byte character excluding the carriage return, line feed, tab or quotation mark characters; and in the second instance a *Character* is any single byte character excluding the carriage return, line feed, tab or apostrophe characters.

Number



A *Digit* is any of the characters '0' to '9'.

The driver value indicates the location and name of the driving process. This is required by the internal sort interface library (ppmsapi.so). It is not required if the internal sort interface is not used.

The phase1 and phase2 values indicate to the PPMSORT driving process (ppmsort) where the phase one and phase two component commands are located. The value is the full or relative path of of the corresponding component. Ordinarily, these values should not be changed for a particular machine or environment once the software has been installed.

The driver_nice, phase1_nice and phase2_nice values are used for adjusting the relative process priorities and correspond to the driver, phase1 and phase2 process priorities, respectively. In each case the process priority is adjusted using the supplied value as the argument in a nice(2) call.

The license string is expected to comprise of a sequence of characters which are valid as hexadecimal. The supplied license key validates against the current machine and hence each machine should have its own license key generated. The key value must be exactly 32 characters long, representing the value of an eight byte key:

```
license = "9312975875F523A17A8B09E8E7E09EF4";  /* license key */
```

It is also possible to supply the license key value directly or indirectly using an appropriate environment variable (CODEMAGUS_RUNTIME_KEY_PPMSORT or CODEMAGUS_KEYFILE). These environment variables supply the value of the license key or the path and file name of the license key file.

The max_records parameter gives the upper bound on the number of records that can be considered for sorting in a single instance of a sort phase process.

The max_bytes parameter gives the upper bound on the number of bytes that can be considered for sorting in a single instance of a sort phase process.

In the following example, the maximum number of records that a single sort phase process can deal with is bounded by 0.1 million and the maximum number of bytes that any sort phase process can deal with is bounded by 40 million:

```
max_records = 100000;    -- maximum number of records per partition
max_bytes = 40000000;    -- maximum number of bytes per partition
```

The driving process spawns a number of phase one and phase two helper processes. A phase one helper process is responsible for sorting a portion of the input file and a phase

two helper process is responsible for merging a number of sorted files together. Because the resource profile of the phase one and phase two processes can be significantly different, depending on the values set by the other resource constraint parameters, the upper bound of the number of each of these process types can be specified and individually controlled. The parameter `max_phase1_procs` provides an upper bound to the maximum number of concurrent phase one processes whilst the `max_phase2_procs` parameter provides the same control for the phase two processes.

In the following example, the number of concurrent phase one processes is limited to 20 and the number of phase two processes is limited to 100:

```
max_phase1_procs = 20;  -- maximum number of helper sort processes
max_phase2_procs = 100; -- maximum number of helper merge processes
```

The `max_files` parameter gives the upper bound of sorted input files that each of the merge processes may read. This number must be less than the number of open file descriptors that the operating system allows per process. The following example limits this number to 100 open input files:

```
max_files = 100;        -- maximum number of input sort streams per process
```

The `stream_buffer_size` parameter is used to supply the value to be used as the size of the stream buffers. The following example sets the size of the stream buffers used to read and write application data and sort work files to 40 kilo-bytes:

```
stream_buffer_size = 40000;   --- stream buffer size
```

The `sortwork` parameter is used to provide the path where `PPMSORT` places its work files. Ordinarily, unless the `verbose` option is used, the work files are removed as the sort progresses. The names of the files are designed not to interfere with one another by including the process id and a sequence number as part of the file name. For example, the file name `sortwork_00021121_00000000` is the first work file to be used by the sort initiated by the driving process `ppmsort` with PID 21121.

The following parameter indicates that `/tmp` should be used for the work files:

```
sortwork = "/tmp";      -- location of temporary sort work files
```

The following is a full configuration file example:

```
-- options verbose;
-- options stable, ascii_as_ebcdic;
   options stable;
   phase1 = "/home/stephen/bin/ppmsqsort";
   phase2 = "/home/stephen/bin/ppmsmerge";

-- license = "9312975875F523A17A8B09E8E7E09EF4";  /* license key */
   max_records = 100000;   -- maximum number of records per partition
   max_bytes = 40000000;   -- maximum number of bytes per partition
   max_phase1_procs = 20;  -- maximum number of helper sort processes
   max_phase2_procs = 100; -- maximum number of helper merge processes
   max_files = 100;        -- maximum number of input sort streams per process
```

```
sortwork = "/tmp";      -- location of temporary sort work files
```

3 Command Line Arguments

All the configuration file parameters can be overridden or supplied as command line arguments. The only exceptions are the license key parameter and the phase one and phase two component paths and names.

Additionally, the sort requirements such as sort key specification, input and output file attributes are exclusively supplied using command line arguments.

3.1 Configuration File Parameters

Command line arguments can be used to override the resource bounds which may or may not have been supplied in a configuration file. The following command line arguments supply or override the indicated resource bound value from the configuration file.

- `--max-records` or `-p`:
  This command line argument overrides the `max_records` configuration file parameter; or supplies a value in the absence of a configuration file setting for this parameter. This value is used to limit the number of records that a phase one process should sort.
- `--max-bytes` or `-b`:
  This command line argument overrides the `max_bytes` configuration file parameter; or supplies a value in the absence of a configuration file setting for this parameter. This value is used to limit the number of bytes that a phase one process should sort.
- `--max-qsort-procs` or `-q`:
  This command line argument overrides the `max_phase1_procs` configuration file parameter; or supplies a value in the absence of a configuration file setting for this parameter. This value is used as a bound to the number of concurrent phase one processes.
- `--max-merge-procs` or `-m`:
  This command line argument overrides the `max_phase2_procs` configuration file parameter; or supplies a value in the absence of a configuration file setting for this parameter. This value is used as a bound to the number of concurrent phase two processes.
- `--stream-buffer-size` or `-s`:
  This command line argument overrides the `stream_buffer_size` configura-

tion file parameter; or supplies a value in the absence of a configuration file setting for this parameter. This value is used as a bound to the number of concurrent phase two processes.

- `--max-files` or `-f`:
  This command line argument overrides the `max_files` configuration file parameter; or supplies a value in the absence of a configuration file setting for this parameter. This value is used as a bound to the number of open files that a phase two process can have open.
- `--sort-work-file-path` or `-w`:
  This command line argument overrides the `sortwork` configuration file parameter; or supplies a value in the absence of a configuration file setting for this parameter. The value is used as the location where the current instance of `PPMSORT` will place intermediate sort work files.

The following command line arguments also pertain to resources, but not in the same manner as the resource bound command line arguments. Instead, these command line arguments adjust the relative priority of the driver process and that of the phase processes. The have the effect of reducing the CPU resource consumption rate in the presence of work with a higher relative priority.

- `--driver-nice` or `-n`
  Set the driver process relative priority using the supplied value as a parameter to the `nice(2)` system call. This command line argument overrides the value of the `driver_nice` configuration file parameter.
- `--qsort-nice` or `-a`
  Set the phase 1 process relative priority using the supplied value as a parameter to the `nice(2)` system call. This command line argument overrides the value of the `phase1_nice` configuration file parameter.
- `--merge-nice` or `-c`
  Set the phase 2 process relative priority using the supplied value as a parameter to the `nice(2)` system call. This command line argument overrides the value of the `phase2_nice` configuration file parameter.

3.2 Configuration File Options

Command line arguments can be used to supply certain options which may or may not have been specified as options in a configuration file. Once an option is supplied the flag value cannot be negated by, for example, a subsequent command line argument. The only exception to this is the use of the `--quiet` command line argument to reset the `verbose` option.

- `--stable-sort` or -t:
  This command line argument flags the same option as `stable` does in a configuration file.
- `--ascii-as-ebcdic` or -e:
  This command line argument flags the same option as `ascii_as_ebcdic` does in a configuration file.
- `--verbose` or -v:
  This command line argument flags the same option as `verbose` does in a configuration file.
- `--quiet` or -u:
  This command line argument resets the same flag that option `verbose` sets in a configuration file. This option is used to negate a configuration file specified `verbose` flag.

3.3 Application Sorting Requirements

The input and output file names and attributes and application level sorting requirements are specified to the sort driving process (`ppmsort`) as command line arguments. There is no configuration file alternative for these command line arguments as the values could be specific for each usage of the `PPMSORT` utility.

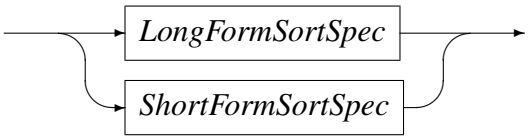
- `--input-file-name` or `-i`:
  This command line argument supplies the name of the unsorted input file.
- `--output-file-name` or `-o`:
  This command line argument supplies the name to which the sorted output data should be written. The file does not have to exist. If it does it will be overridden and if it does not it will be created. If the input file is empty then the output file will still be opened according to the default or supplied output open file mode parameter. This will have the effect of creating an empty output file (if this is what is implied by the output open mode parameter) for an empty input file or if an error should occur before the output phase of the sort.
- `--record-length` or `-l`:
  This command line argument supplies the record length if the records in the input file have a fixed length. If the file is a Unix (for example) text based file then this length needs to take into account the record terminating new line character at the end of each record.
- `--use-rdwio` or `-d`:
  This command line argument indicates that the records in the input and output files are variable in length and that the length of each record is described by a record descriptor word (the record descriptor word is a four byte entity which

comprises a two byte big-endian length followed by a two byte low value field; the length includes the length of the four byte RDW).

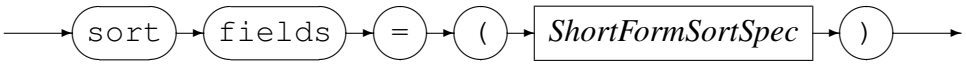
- `--drop-duplicates` or `-r`:
  This command line argument indicates that records with a key value already encountered should be dropped as duplicates. Used in conjunction with the `--stable-sort` command line argument or the `stable` configuration file option, the `--drop-duplicates` argument will result in the first record with a duplicate key value to be kept in the output file.

- `--sort-key` or `-k`:
  This command line argument supplies the application sort key specification. The syntax and semantics of this field are described below:

  SortSpec

  LongFormSortSpec

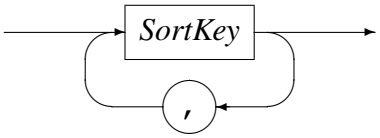
  

  ShortFormSortSpec

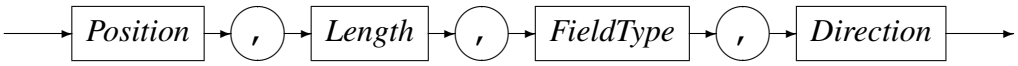
  

  The *LongFormSortSpec* and the *ShortFormSortSpec* have the same meaning.

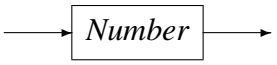
  SortKeyList

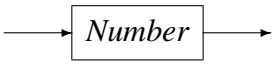
  SortKey

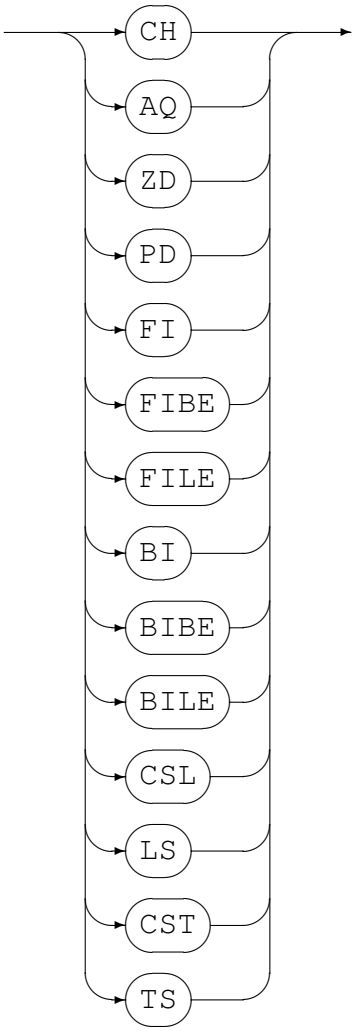
  Position

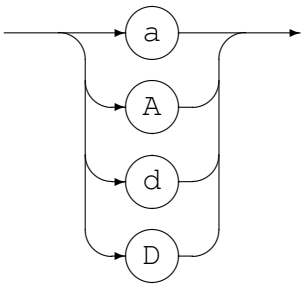
  Length

FieldType



Direction



The sort key specification comprises a list of fields with sub-fields *Position*, *Length*, and *FieldType*, together with a *Direction* indicator of whether the corresponding record field should be sorted in ascending or descending order. Each such group sub-field of the sort key specifies the major key items starting on the left, possibly, followed by the key to be sorted within duplicates of the keys specified to its left;

with each item corresponding to a record field.

The *Position* sub-field indicates the first byte position containing the sort field being described. This is the position of the data within the record where the first position is one. RDW prefixes are not counted as part of the data (hence the first byte of a variable length record also has a *position* of one).

The *Length* sub-field indicates the length in bytes of the record field being described.

The *FieldType* sub-field describes the type of the corresponding record field. The type field keywords can be specified in either upper or lower case and have the following definitions:

`CH`: Character data. The record field is assumed to be an alphanumeric character field. The ordering of the data within these fields is lexicographical within the ASCII or EBCDIC collating sequences.

If all character data is to be ordered according to the EBCDIC collating sequence, then the `ascii_as_ebcdic` configuration file option or the `--ascii-as-ebcdic` command line argument should be used.

`AQ`: Alternate collating sequence character data. The behaviour of sorting on this field is the same as on `CH` except that when *FieldType* is `AQ` the data is interpreted as being in the ASCII collating sequence as 7-bit data. The high order bit is still used to sort the data, but when the data is to be ordered according to the EBCDIC collating sequence then only the byte values below 128 are translated to the corresponding EBCDIC values. All other byte values retain their value. Note that in this case the use of `AQ` causes multiple mappings to the same value in the collating sequence.

If all character data is to be ordered according to the EBCDIC collating sequence, then the `ascii_as_ebcdic` configuration file option or the `--ascii-as-ebcdic` command line argument should be used.

`ZD`: Zoned decimal. The data is expected to be numeric character data where the sign of the number is over-punched into the last (right-most) digit position. For ASCII based numeric character data a negative sign is indicated by all the 0x70 bits being on in the high order nibble of the byte. Positive numbers are indicated by the absence of a sign in the high-order nibble (i.e. the high-order nibble should only have the 0x30 bits on).

`PD`: Signed packed decimal. The data is expected to comprise of decimal digits, two per byte. The values corresponding to the decimal digits are the hexadecimal values 0x0 to 0x9. A sign is expected to occupy the low-order nibble of the last byte. A sign position value of 0xF or 0xC indicates a positive number, all other values are treated as negative sign values.

`FI`: Signed binary. The data is expected to represent a numeric item as a signed binary number. The endian-ness of the number is determined by the architecture of the host machine.

`FIBE`: Signed binary. The data is expected to represent a numeric item as a signed binary number. This number is represented in a big-endian byte order (most significant byte first), and not in a byte order determined by the architecture of the host machine.

`FILE`: Signed binary. The data is expected to represent a numeric item as a signed binary number. This number is represented in a little-endian byte order (least significant byte first), and not in a byte order determined by the architecture of the host machine.

`BI`: Unsigned binary. The data is expected to represent a numeric item as an unsigned binary number. The endian-ness of the number is determined by the architecture of the host machine.

`BIBE`: Unsigned binary. The data is expected to represent a numeric item as an unsigned binary number. This number is represented in a big-endian byte order (most significant byte first), and not in a byte order determined by the architecture of the host machine.

`BILE`: Unsigned binary. The data is expected to represent a numeric item as an unsigned binary number. This number is represented in a little-endian byte order (least significant byte first), and not in a byte order determined by the architecture of the host machine.

`CSL` or `LS`: Sign leading separate. The data is expected to comprise of numeric data with a leading character sign. If the leading character is a minus sign then the number is treated as a negative number. All other signs result in the number being treating as a positive number.

`CST` or `TS`: Sign trailing separate. The data is expected to comprise of numeric data with a trailing sign character. If the trailing character is a minus sign then the number is treated as negative number. All other signs result in the number being treated as a positive number.

The *Direction* sub-field indicates whether or not that portion of the record key should be sorted within the file (for the major key or first record key field), or the record key should be sorted within the duplicates of the major portions of the record key (for minor or subsequent record key fields) in an ascending or descending key sequence manner. The characters `A` or `a` indicate that the portion of the key should be sorted in an *ascending* sequence; whilst the characters `D` or `d` indicate that the portion of the key should be sorted in a *descending* sequence.

4 Sort Examples

4.1 Sorting ASCII Print Data

Given the following contents (assumed to be in the `sortchars.dat`) containing ASCII text data:

```
MMMMMMMMMM 00000000
%%%%%%%%%% 00000001
UUUUUUUUUU 00000002
CCCCCCCCCC 00000003
8888888888 00000004
rrrrrrrrrr 00000005
8888888888 00000006
XXXXXXXXXX 00000007
iiiiiiiiii 00000008
BBBBBBBBBB 00000009
%%%%%%%%%% 00000010
pppppppppp 00000011
KKKKKKKKKK 00000012
oooooooooo 00000013
&&&&&&&&&& 00000014
(((((((((( 00000015
ffffffffff 00000016
uuuuuuuuuu 00000017
gggggggggg 00000018
dddddddddd 00000019
```

The following command sorts the file using the ten characters at the front of each record:

```
[stephen@nomad sorttest]$ ppmsort --input-file-name=sortchars.dat \
        --output-file-name=sortchars.txt \
        --record-length=20 \
        --sort-key="SORT FIELDS=(1,10,CH,A)"
```

The output of the sort includes a summary report of the processing:

```
[ppmsort] $Id: ppmsort.c,v 1.23 2010/04/07 18:50:42 hayward Exp $
Copyright (c) 2007 by Code Magus Limited. All rights reserved.
mailto:stephen@codemagus.com, http://www.codemagus.com.
Total bytes input to sort process      = 400.
Total records input to sort process    = 20.
Total bytes output from sort process   = 400.
Total records output from sort process = 20.
```

The output file of this command is:

```
%%%%%%%%%% 00000001
%%%%%%%%%% 00000010
&&&&&&&&&& 00000014
(((((((((( 00000015
8888888888 00000004
8888888888 00000006
```

```
BBBBBBBBBB 00000009
CCCCCCCCCC 00000003
KKKKKKKKKK 00000012
MMMMMMMMMM 00000000
UUUUUUUUUU 00000002
XXXXXXXXXX 00000007
dddddddddd 00000019
ffffffffff 00000016
gggggggggg 00000018
iiiiiiiiii 00000008
oooooooooo 00000013
pppppppppp 00000011
rrrrrrrrrr 00000005
uuuuuuuuuu 00000017
```

In this example the collating sequence is clearly visible as ASCII. This can be seen in that the special characters (at least the ones shown), precede the numerics, which precede the upper case characters, which precede the lower case characters.

4.2 Dropping Duplicates

The following command includes the `--drop-duplicates`:

```
[stephen@nomad sorttest]$ ppmsort --input-file-name=sortchars.dat \
        --output-file-name=sortchars.txt \
        --record-length=20 \
        --sort-key="SORT FIELDS=(1,10,CH,A)" \
        --drop-duplicates
```

This produces the summary report:

```
[ppmsort] $Id: ppmsort.c,v 1.23 2010/04/07 18:50:42 hayward Exp $ "
Copyright (c) 2007 by Code Magus Limited. All rights reserved.
mailto:stephen@codemagus.com, http://www.codemagus.com.
Total bytes input to sort process      = 400.
Total records input to sort process    = 20.
Total bytes output from sort process   = 360.
Total records output from sort process = 18.
Total duplicate output records dropped = 2.
```

The output file now excludes records where the composite sort key value is duplicated:

```
%%%%%%%%%% 00000001
&&&&&&&&&& 00000014
(((((((((( 00000015
8888888888 00000004
BBBBBBBBBB 00000009
CCCCCCCCCC 00000003
KKKKKKKKKK 00000012
MMMMMMMMMM 00000000
UUUUUUUUUU 00000002
XXXXXXXXXX 00000007
```

```
ddddddddddd 00000019
fffffffffff 00000016
ggggggggggg 00000018
iiiiiiiiii 00000008
ooooooooooo 00000013
ppppppppppp 00000011
rrrrrrrrrr 00000005
uuuuuuuuuu 00000017
```

4.3 Sorting ASCII Data in an EBCDIC Sequence

The following command illustrates how ASCII (or ISO-8) data can be sorted as though it is ordered in the sequence of EBCDIC characters.

```
[stephen@nomad sorttest]$ ppmsort --input-file-name=sortchars.dat \
        --output-file-name=sortchars.txt \
        --record-length=20 \
        --sort-key="SORT FIELDS=(1,10,CH,A)" \
        --ascii-as-ebcdic
```

The output file can be seen as adhering to an EBCDIC collating sequence in which the special characters (shown) precede the lower-case characters, which precede the upper-case characters and which precede the numeric characters.

```
(((((((((( 00000015
&&&&&&&&&& 00000014
%%%%%%%%%% 00000001
%%%%%%%%%% 00000010
ddddddddddd 00000019
fffffffffff 00000016
ggggggggggg 00000018
iiiiiiiiii 00000008
ooooooooooo 00000013
ppppppppppp 00000011
rrrrrrrrrr 00000005
uuuuuuuuuu 00000017
BBBBBBBBBB 00000009
CCCCCCCCCC 00000003
KKKKKKKKKK 00000012
MMMMMMMMMM 00000000
UUUUUUUUUU 00000002
XXXXXXXXXX 00000007
8888888888 00000004
8888888888 00000006
```