Sort Solution

Homepage at http://www.mwlabs.de

You must read the information about License and Copyright before you use this software!

Welcome to Sort Solution & Tools!
Sort Solution is a library with powerful routines for sorting and merging external files for Windows 95 and Windows NT. With it's unique algorithms and parallel sorting techniques, Sort Solution is able to sort files of any size with an absolute minimum of RAM at an enormous speed. The maximum file size that can be sorted with Sort Solution is only limited by the Operating System. The maximum file size for Windows 95 is 2 GB (or up 8 GB with the "b" version), for Windows NT the file size is theoretically unlimited.

A variety of different input formats and key types distinguishes Sort Solution from the other sort utilities and libraries that have been available on the market for UNIX and Mid-Range systems. The performance of Sort Solution competes with the fastest sort tools available today or even exceeds their capabilities and power.

Programmers can integrate Sort Solution into their applications with only a few simple functions calls. All the required libraries for C/C++ and Visual Basic are included in the package. Sort Solution supports all 32-Bit Windows programming environments which are able to use functions from external DLL's.

Sort Solution is distributed as Shareware. You can use the library and the supplied tools without any risk or cost to see if they will fit your needs. When you decide to continue using Sort Solution, you have to pay a small Shareware fee (see Shareware and Registration of Sort Solution). Paying the fee will entitle you for discounts on future versions and support direct from the author.

The free trial version of Sort Solution has the complete functionality of the licensed version of Sort Solution with one minor limitation.

A note to all native speakers:
I'm a German programmer and English is not my primary language. I've done my best while writing this manual, so please be kind and forgiving when you find some strange expressions or grammar J

License and Copyright

Sort Solution and the Sort Solution Tools

SHAREWARE

Shareware distribution gives users a chance to try software before buying it. If you try a Shareware program and continue using it, you are required to register it (or purchase the licensed version as in the case of Sort Solution).

Copyright laws apply to both Shareware and retail software, and the copyright holder retains all rights, with a few specific exceptions as stated below. Shareware authors are accomplished programmers, just like retail authors, and the programs are of comparable quality. (In both cases, there are good programs and bad ones!)

Shareware is a distribution method, not a type of software. You should find software that suits your needs and pocketbook, whether it's retail or Shareware. The Shareware system makes fitting your needs easier, because you can try before you buy. And because the overhead is lower, prices are lower also. Shareware has the ultimate money-back guarantee -- if you don't use the product, you don't pay for it.

LIMITED WARRANTY AND DISCLAIMER OF WARRANTY

THIS SOFTWARE AND ACCOMPANYING WRITTEN MATERIALS (INCLUDING INSTRUCTIONS FOR USE) ARE PROVIDES "AS IS" WITHOUT WARRANTY OF ANY KIND. FURTHER, Mario M. Westphal DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF USE, OF THE SOFTWARE OR WRITTEN MATERIALS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. IF THE SOFTWARE OR WRITTEN MATERIALS ARE DEFECTIVE YOU, AND NOT Mario M. Westphal OR ITS DEALERS, DISTRIBUTORS, AGENTS, OR EMPLOYEES, ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

THE ABOVE IS THE ONLY WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, THAT IS MADE BY Mario M. Westphal, ON THIS PRODUCT. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY Mario M. Westphal, ITS DEALERS, DISTRIBUTORS, AGENTS OR EMPLOYEES SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY AND YOU MAY NOT RELY ON ANY SUCH INFORMATION OR ADVICE.

NEITHER Mario M. Westphal NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION OR DELIVERY OF THIS PRODUCT SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL OR INCIDENTAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE SUCH PRODUCT EVEN IF Mario M. Westphal HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

In other words: THERE IS NO GUARANTEE! **YOU USE THIS SOFTWARE PRODUCT ON YOUR OWN RISK!**

IF YOU DON'T LIKE THIS, SIMPLY DON'T USE Sort Solution!

IN STATES/COUNTRIES WHERE THESE RESTRICTIONS ARE ILLEGAL, YOU ARE NOT ALLOWED TO USE Sort Solution!

ACKNOWLEDGMENT

BY USING THE SHAREWARE VERSION OF Sort Solution YOU ACKNOWLEDGE THAT YOU HAVE READ THIS LIMITED WARRANTY, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS' TERMS AND CONDITIONS. YOU ALSO AGREE THAT THE LIMITED WARRANTY IS THE COMPLETE AND EXCLUSIVE STATEMENT OF AGREEMENT BETWEEN THE PARTIES AND SUPERSEDE ALL PROPOSALS OR PRIOR AGREEMENTS, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN THE PARTIES RELATING TO THE SUBJECT MATTER OF THE LIMITED WARRANTY.

Copyright

License

You may use the shareware version of Sort Solution for a 30 day trial period. If you would like to continue to use the product after the 30 day trial period, you are required to purchase the licensed version of Sort Solution. See Shareware and Registration of Sort Solution for further details an how to license your copy of Sort Solution.

If you want use Sort Solution in your applications, you need to buy a licensed version. Then you are allowed to distribute Sort Solution together with your applications without an additional license fee (no royalties).

You are NOT allowed to develop an application with Sort Solution which main purpose can be described as »Sorting files«. This means also, that you are not entitled to create another Sort-Utility which basically has the same functionality as Sort Solution.

You are not allowed to distribute the header files and libraries that are part of the Sort Solution package to your customers without an explicit written agreement of the author.

Thank You!

I wish to thank the following people who assisted me in testing Sort Solution:

- Heinz Bartkewitz
- Reinhard Iben
- Michael Kanzler
- Marc Müller
- Stefan Schneider
- Thomas Vaughan

I also wish to thank the customers who contributed tips and improvements and reported bugs.

Trademarks

Shareware and Registration of Sort Solution

See also License and Copyright

You may use the shareware version of Sort Solution for a 30 day trial period. If you would like to continue to use Sort Solution after the 30 day trial period, you are required to purchase the licensed version of Sort Solution.

You should carefully read the terms and conditions stated under License and Copyright before using this software.

Please read also the comments on Hardware Requirements and Limits.

The Author

Mario M. Westphal
Pestalozzistraße 6-12
61250 Usingen
Germany

Email: *support@mwlabs.de*
Homepage at http://www.mwlabs.de

Restrictions in the Shareware version

The free available shareware version of Sort Solution comes with all functions enabled and the complete manual, libraries, and samples. The only limitation is that **some of the records in the output file are skipped by intention**. This means, that all output files created with the unlicensed version have 1 to 10 records less than the input file. The records that are skipped are selected randomly.

This restriction of course **does not apply** to the licensed version.

Additionally, the unlicensed version of Sort Solution prints an informative message to the screen on every run. The licensed version does not print this message.

Distributing Sort Solution

The **unregistered** version of Sort Solution can be distributed freely under the fact that the complete package (sosodis.exe) remains unchanged and complete. Also there should be a notice that the newest version of this Software can be found on the authors homepage at http://www.mwlabs.de

If you want to distribute Sort Solution together with your own applications, you are required to license a copy of Sort Solution. If you work in a development team, every person that works on the project that incorporates the Sort Solution technology also needs a licensed version of Sort Solution.

Licensing Sort Solution

The distribution file *sosodis.exe* contains a licensed version of SORTSOL.EXE and SORTSOL.DLL in a **password protected** ZIP file. When you register your copy of Sort Solution, you get the password for this file directly by the author via email. These two files replace the unlicensed versions which are installed when you setup Sort Solution for evaluation purposes.

One licensed copy of Sort Solution may either be used by a single person who uses the software personally on one or more computers, or installed on a single workstation used non-simultaneously by multiple people, but not both.

You may access the licensed version of Sort Solution through a network, provided that you have obtained individual licenses for the software covering all workstations that will access the software through the network. For instance, if 5 different workstations will access Sort Solution on the network, each workstation must have its own Sort Solution license, regardless of whether they use Sort Solution at different times or concurrently.

If Sort Solution is used in a project with multiple developers, a separate license must be bought for every developer involved in the project. Say, for a project which is developed by five persons, and all those persons use or compile source code that contains calls to the Sort Solution API or ActiveX, five separate copies of Sort Solution have to be licensed.

How to Obtain a License

The install directory of Sort Solution contains a file named **REGISTER.WRI**. This file contains the latest info on how to register your copy Sort Solution.

Hardware Requirements and Limits

SORTMEM THREADS MERGEMEM Performance Tuning

- 486 INTEL or compatible
- Windows 95 or Windows NT 4.x
- 16 MB of RAM or more (always better: more!)
- At least one hard disk
- Optionally a mouse or a compatible pointing device

Limits

The following table lists the limits of Sort Solution. Please consider this table when you create a profile.

Maximum file size:	Limited only by the Operating System
Maximum record length	65535 byte (64K). For file formats that use record delimiters, this includes the size of the record delimiter
Maximum number of fields per record	Unlimited, as long as the sum of the size of all field does not exceed the record limit of 64K
Keys per sort	1 - 64
Key length	1 - 65535 bytes
Threads	1 - 64
Pre-Merge caches	1 - 128
Temporary files (disks)	Up to 64

Memory Usage

Sort Solution needs memory for the sort, the merge and for it's internal data structures and caches. Normally, the internal memory usage is about 256 Kbytes, which should be not critical at all.

You can get a raw estimate of the memory footprint of Sort Solution with the following formula:

```
Threads * Memory per Thread + Caches * Memory per Thread + 64K + Threads * ((Memory per Thread / Record Length)*14)
```

Example

Threads	4
Caches	8
Memory per thread	512 K
Record length	64 Byte

```
4 * 512 * 8 * 512 + 64 + 4 * ((512/64)*14) = 6656K, approx. 7 MB
```

You should avoid to use to much memory per thread, especially when you sort files that contain a large number of very small records (less than 16 byte). In this case, Sort Solution needs more memory for it's internal data structures than for the actual file data.
If the Operating System is unable provide the required amount of memory for the sort, it must swap main memory to and from the disks which results in a significant performance penalty.

Quick Start and Tutorial

This chapter describes how to configure and use Sort Solution and SORTSOL.EXE, the standalone sort utility which is part of the Sort Solution package. SORTSOL.EXE is a command line tool based on the Sort Solution technology, which allows you to sort any kind of file directly from the Windows 95 / Windows NT command line.

This chapter presents most of the concepts of Sort Solution and applies them to some real-world examples. After you have studied the tutorial, you should be able to apply your new knowledge to your actual sort problems.

Tip You find more information on the technical backgrounds of Sort Solution under the topic Sort Solution Technical Backgrounder.

Note This tutorial assumes that you have installed Sort Solution in the folder »c:\sortsol«. .If you have installed Sort Solution into another directory, please replace all references to »c:\sortsol« with the folder you have chosen during the installation.

Outline

The tutorial is divided into six sections:

What do I need to sort a file with Sort Solution?

To sort a file with Sort Solution, you first have to gather some information about the format of the file and the format of the records contained within.

- The name of the file to be sorted (the *input file*)
- The name of the *output file*, if different from the input file (You can use the same file for input and output)
- The *format* of the records, e.g. delimited or fixed length
- The criteria after which the file should be sorted

With this information at hand, you create a simple text document containing statements from the the Sort Solution Script Language. This so-called *profile* describes *what* to sort and *how* to sort it.

The Sample File

The folder »\Samples\Tutorial« of your Sort Solution installation contains a file called »sales.txt«. This file will serve as the input file during this tutorial.

»sales.txt« is a file with a fixed record layout, all records have the same size of 94 byte.

Note: You can, of course, sort files with variable length records with Sort Solution, but a file with a fixed record length is much easier to read and therefore serves better for this tutorial. You find a list with all files types supported by Sort Solution here.

File Format of »sales.txt«

Every record in »sales.txt« consists of 8 *fields*, as described in the following table.

Nr	Name of the Field	Starts at Offset*	Length	Data Type
1	Product	0	12	Text
2	Distribution	12	12	Text
3	City	24	20	Text
4	Date	44	8	Date
5	Time	52	8	Time
6	Amount	60	10	Integer
7	Value	70	22	Float
8	Delimiter	92	2	Carriage Return / Linefeed (0x0D,0x0A)
			94 Byte	

**The term Offset means the distance in byte from the beginning of the record.*

Here are some sample records from the file »sales.txt«:

- The character t stands for the record delimiter (Carriage Return / Linefeed)
- The character | was inserted here to make it easier to distinguish individual fields within the records. It is *not* part of the file itself

```
Product 1   |Direct      |Berlin              |31.03.93| 0:00:00|62        |1564,26
t
Product 1   |Direct      |Los Angeles         |30.04.93| 0:00:00|117       |2951,91
t
Product 1   |Reseller    |Tokyo               |31.03.93| 0:00:00|50        |1261,50
t
Product 1   |Reseller    |Buenos Aires        |30.06.93| 0:00:00|4         |100,92
t
Product 1   |Reseller    |Oslo                |30.04.93| 0:00:00|33        |832,59
t
Product 1   |Dist        |Berlin              |30.06.93| 0:00:00|13        |327,99
t
...
```

Step 1: Creating a Profile

To sort the file »sales.txt« you first have to create a *profile* which contains the commands for the sort utility SORTSOL.EXE.

1. Open the Windows 95 / Windows NT command prompt
   (Open the »Start« menu, click on »Programs« and then on »Command Prompt«)
2. Change to the directory »c:\sortsol\samples\tutorial«
   Note: Please replace »*c:**sortsol*« with the directory where you have installed the Sort Solution package
3. Enter the following statement at the command prompt:

   c:\sortsol\bin\sortsol -ccity

This command executes SORTSOL.EXE and creates a default profile with initial settings called »city.ssp«. You can then open the profile with any text editor, e.g. the Windows Notepad, by typing

notepad city.ssp

at the command prompt.

The file generated by SORTSOL.EXE will look like this:

```
01 ; Generated SORTSOL profile
02 ; Search for TODO to complete this file
03
```

```
04 INPUTFILE(TODO: Insert input filename)
05 OUTPUTFILE(TODO: Insert output filename)
06 FILETYPE(TODO: Insert a file type specifier and parameters)
07
08 KEY(TODO: Insert a key type specifier and arguments)
```

The line numbers at the beginning of the lines are *not* part of the file, but are inserted here for explanation purposes.

Line 01 and 02 do contain two *comments*, explaining what to do with the rest of the file. SORTSOL.EXE ignores all lines beginning with a semicolon (»;«). This gives you the opportunity to insert comments into you profiles for a better understanding of what is the purpose of the profile.

Line 04 contains a INPUTFILE statement which describes the file to be sorted. This statement is one of the commands available in the Sort Solution Script Language.

Line 05 contains an OUTPUTFILE statement. This statement is used to name an output file for the sorted data. If you don't use an OUTPUTFILE statement in your profile, Sort Solution will overwrite the input file with the sorted data.

Line 06 contains a FILETYPE statement. The FILETYPE statement describes the format of the input file.

Line 08 contains a KEY statement. KEY statements are used to define which part of the record is used as the *sort key*.

All statements in a default profile initially do contain only the text *TODO:* and a comment explaining what has to be done to complete the statement.

This leads us to the next step:

Step 2: Editing the Profile

Open the profile »city.ssp« with your favorite text editor, e.g. the *Windows Notepad* or *Wordpad*. Then apply the changes (printed in **bold**) as shown in the next paragraph and save the file.

```
01 ; Generated SORTSOL profile
02 ; Search for TODO to complete this file
03
04 INPUTFILE(c:\sortsol\samples\tutorial\sales.txt)
05 OUTPUTFILE(c:\sortsol\samples\tutorial\sorted.txt)
06 FILETYPE(FIXED,94)
07
08 KEY(String,ASC,0,24,20)
```

Line 04 has been changed to contain the name and path of the file to be sorted, in our case »c:\sortsol\samples\tutorial\sales.txt«. Again, this assumes that you have installed Sort Solution into »c:\sortsol«. If you have installed it into another directory, please exchange »c:\sortsol« with the name of your install directory.

Line 05 contains the name of the output file. Sort Solution writes the sorted data from the input file to the file » c:\sortsol\samples\tutorial\sorted.txt «.

Line 06 describes the input file as *FIXED,94* which describes a fixed length file (each record has the same length) with a record length of 94 bytes.

The KEY statement in line 08 describes which part of each record contains the sort criteria and which data type should be used to interpret the data in the record.
In this case, the key type is *String* (ASCII String with a fixed or variable length), the sort order is ASCending), and the part of the record, which is used as the sort criteria starts at offset 24 and has a length of 20 byte. From the file format description above you know that this is the record field »*City*« which holds the name of the city in each record.

This profile sorts the file »sales.txt« after the »City« field in ascending order and stores the sorted output into the file »sorted.txt«

Tip If you want to sort the file in descending order instead, replace the ASC with DESC in line 04.

Step 3: Running the Sort

To run the sort, enter the following command at the command prompt:

```
c:\sortsol\bin\sortsol city.ssp
```

Please replace the path »c:\sortsol« with the directory to where you installed the Sort Solution package.

As far as there is no typo or syntactical error in the profile, SORTSOL.EXE should finish in a few seconds. If there is any error reading the profile or executing the sort, SORTSOL.EXE will print an appropriate error message on the screen. In this case, open you profile and make sure that everything looks *exactly* like in the example profile above.

Please open now the file »sorted.txt« with a text editor and take a look on the sort order created by Sort Solution. The file should be sorted after the city field, the order of the records for each city is not defined and should be the same as in the input file.

Tip: If you use Notepad, change to a fixed font. This makes it much easier to read the file.

During the sort, SORTSOL.EXE will display a bunch of messages on the screen:

```
01 Sort Solution Version 1.2.2
02 Copyright (C) 1998 Mario M. Westphal
03
04 Sort(OP)..: 100%
05
06
07 Input file                    : c:\sortsol\samples\tutorial\SALES.TXT
08 Output file                   : c:\sortsol\samples\tutorial\SORTED.TXT
09 Log file                      :
10 Input Filesize                : 89,770 Bytes
11 Records processed             : 955
12 Records filtered:             : 0 (> 0 in the unlicensed version)
13 Time to complete              : 0s (Sort: 0s, Merge: 0s)
14   Avg. block time for load    : 31 ms
15   Avg. block time for sort    : 47 ms
16   Avg. block time for pre-merge : 0 ms
17 Number of runs                : 0
17 Cache per run                 : 0 KB
```

Line 07	Name of the input file
Line 08	Name of the output file
Line 09	Name of the logfile (not used in this example)
Line 10	Size of the input file in byte
Line 11	Number of records processed
Line 12	Number of records filtered. Unless you as Filter, this will be 0, except you use the unlicensed version of Sort Solution which will always skip some of the records in the output file. Please see **Limitations in the unregistered version** for more information about this issue.
Line 13	Time for the sort in seconds. For files of this size, the sort should take only a few seconds. Remember that Sort Solution is prepared to sort files of several gigabytes with millions or even billions of records

The statistical numbers in line 14 to 17 can be used to optimize the performance of Sort Solutions for very big files or complex sort jobs. More information on this subject can be found under the topic Performance Tuning.

Other Sorts

If you take a closer look at the file »sorted.txt« you might notice that is virtually of no use to sort only after the field »City«. It would make more sense to sort after »Distribution« and »City«. This would allow for a better overview over the distribution channels per city.

To include an additional sort criteria in your profile, you simply add a second KEY statement (**bold**):

```
INPUTFILE(c:\sortsol\samples\tutorial\sales.txt)
OUTPUTFILE(c:\sortsol\samples\tutorial\sorted.txt)
FILETYPE(FIXED,94)

; Distribution
KEY(String,ASC,0,12,12)

; City
KEY(String,ASC,0,24,20)
```

The new KEY statement is added before the KEY statement for »City«. Again, the sort key is of type String, the sort order is ascending. From the file format description you get the offset and length of the »Distribution« field in each record:

- Offset from the beginning of the record: 12 byte
- Length 12 byte

Save this profile under the name »dist_city.ssp«.

Sequence of KEY statements

The sequence of KEY statements in profiles defines the *sort sequence*, the order in which the keys are applied. In the above example, the sort uses the sequence *Distribution, City*. The resulting file will have a Grouping over »Distribution« and within each distribution channel a sequence over all cities.

Please execute the new profile with the following command:

```
c:\sortsol\bin\sortsol dist_city
```

After the sort has completed, please load the resulting file »sorted.txt« into Notepad and have a close look at the sequence of the records. There are now two groupings in the file: The first grouping is over »Distribution« and the second grouping is over »City«.

Each key has an order of precedence over from top to bottom, depending on the position of the KEY statement in the profile. Each key serves as a tie-breaker for its immediate predecessor in the profile.

Sort after Product, Distribution, City

This sequence will allow you to see the values by Product per Distribution Channel per City.

But be aware, there's a catch in sorting the Product:

The field Product consists of the string "Produkt" (German for "Product") followed by a number between 1 and 10, e.g. "Produkt 1", "Produkt 2", ..., "Produkt 10".

If you sort this field with the key type »String«, you will notice a problem: the key type »String« uses the ASCII values of the characters in the key field to compare the two keys. For example, the character "A" has the ASCII code 64, "B" has 65 etc.
If Sort Solution performs a comparison between to keys of type »String«, it compares the ASCII codes of both strings character by character until the end of one or both strings is reached or there is a mismatch:

```
ABBA        ASCII codes 65 66 66 65
ABBC        ASCII codes 65 66 66 67
```

The string "ABBC" in this sample is »greater« because the ASCII code of it's fourth character is numerical greater than the ASCII code of the second string ("C" (67) versus "A" (65)).

The problem jumps in when you do compare strings that contain numbers:

```
A3          ASCII codes 65 51
A1          ASCII codes 65 49
A2          ASCII codes 65 50
A10         ASCII codes 65 49
```

Since the key type »String« compares all of these strings on a *per character* basis (byte per byte) and *not* on the numerical value, you will get this sequence:

```
A1
A10
A2
A3
```

The same thing will happen with the »Product« field in »sales.txt«.

Especially when you try to sort fields that contain numbers, this can be a problem with an ordinary sort:

```
1
3
10
11
21
9
2
8
15
```

If you sort these records with the key type »String« or »Generic«, you will get the following sequence after the sort:

```
1
10
11
15
2
21
3
8
9
```

Probably this is not what you expected.

Note This problem is not specific to Sort Solution. When you create folders or files on your hard disk that consist only of numbers, or contain numbers, you will face the same problem when you view the files in the Windows Explorer.
If you have, say, files that are named "file1.txt", "file2.txt", ... "file10.txt", "file 11.txt", you will get a wrong sequence when these files are sorted in the Explorer window. Try it out!

The Key Type IntS

Sort Solution is aware of this problem and hence supplies a special key type for fields that contain (or consist of) numbers. Instead using the key type »String«, simply use the key type IntS (Integer String) instead.

The key type »IntS« interprets the content of the field as a *numerical value* (without decimal places) and performs the comparison based on the real numerical value of the field.

To apply this to the problem of sorting the file »sales.txt« after the field »Product«, you have to change the KEY statement in your profile:

```
KEY(IntS,ASC,0,8,2)
```

Since each field starts with the text "Produkt ", we start sorting the field at offset 8, which means that Sort Solution will interpret the bytes between character position 9 and the rest of the field as a numerical value.

```
--------xx   // Column 9 (Offset 8) is the key
Produkt 2    // Numerical value: 2
Produkt 1    // Numerical value: 1
Produkt 3    // Numerical value: 3
...
Produkt 10   // Numerical value: 10
```

The sort order will then look like this:

```
Produkt 1
Produkt 2
Produkt ...
Produkt 10
```

```
Produkt 11
...
```

Here is the complete profile:

```
INPUTFILE(c:\sortsol\samples\tutorial\sales.txt)
OUTPUTFILE(c:\sortsol\samples\tutorial\sorted.txt)
FILETYPE(FIXED,94)

; Produkt
KEY(IntS,ASC,0,8,2)

; Distribution
KEY(String,ASC,0,12,12)

; City
KEY(String,ASC,0,24,20)
```

Save this profile under the name »Produkt_Dist_Stadt.ssp« and run the sort using the command line:

c:\sortsol\bin\sortsol produkt_dist_stadt

Control the result with the Windows Notepad. The resulting file will look something like this:

```
Produkt 1
      Dist
            Berlin
            Boston
            Buenos Aires
            ...
            Tokyo
      Direct
            Berlin
            Boston
            Buenos Aires
            ...
            Tokyo
      Reseller
            Berlin
            Boston
            Buenos Aires
            ...
            Tokyo
Produkt 2
      Dist
            Berlin
            Boston
            Buenos Aires
            ...
            Tokyo
      ...
...
```

Note The key type »IntS« cannot be used to sort fields containing non-numerical values. Each character in the field must be a valid digit in the range 0..9. You can use the *key offset* and *key length* arguments in the key definition to extract exactly the right portion of the field.
You find more information on key types that sort numerical values under the topic KEY: Numbers as Strings

How to sort Date and Time fields

The file format description above also contains a date field. Field 4 (starting at offset 44) contains a date in the format DD.MM.YY. Two digits for *Day*, two digits for *Month* and two digits for the *Year* (without a century).

Sort Solution implements the key type Date to sort fields containing dates in any format. The key type »Date«

takes a *mask* that describes the format of the date. During the sort, Sort Solution applies this mask to the date field and interprets the content of the field according to the mask.

This unique feature allows you to sort files containing dates in any format without the need to reformat the file before sorting. Additionally, Sort Solution also supplies the key type Time, which works for fields containing time values. You can of course use both keys in one profile to sort after date *and* time simultaneously.

To sort the file »sales.txt« after the date field, you only need to add an additional KEY statement to your profile:

```
KEY(Date,ASC,0,44,"DD.MM.YY")
```

For each field containing a date, Sort Solution extracts the date value according to the mask given in the key definition, converts it into a scalar date representation (a *Julian Date*) and uses this exact representation for the comparison.

More information about the different key types supported by Sort Solution can be found under the topic Key Types.

What's next?

The folder »\Tutorial« of your Sort Solution installations contains additional profiles which can be used to sort »sales.txt« after various criteria.

You should use these profiles as a starting point, modify them to get used to the different concepts of Sort Solution and the Sort Solution profiles.

Once you've learned how to create profiles, you are able to sort nearly any file, even with the most complex sort criteria. Additionally, if you're a programmer and you want to use Sort Solution in your applications, your the half way through when you know how to built profiles. The Sort Solution API utilizes an interface which relies basically on profiles too. More on this subject can be found under Incorperating Sort Solution into Your Applications.

After you have studied the tutorial, you should read the following topics to get more and deeper knowledge about the concepts and features of Sort Solution.

Sort Solution Technical Backgrounder

Supported File Types

Keys

The Sort Solution Script Language

Incorperating Sort Solution into Your Applications

More Examples

Sort Solution Technical Backgrounder

Sort Solution is a 32-Bit Sort Library with extremely powerful sort routines for Windows 95 and Windows NT 4.x. With the tools included in the package, Sort Solution can be immediately utilized as a general purpose sorting tool. Programmers can include the Sort Solution technology into their applications with just a few simple function calls.

Some of the most important features are:

- Maximum file size only limited by the Operating System (biggest file sorted yet: 20.3 Gigabytes!)
- Sort millions or even billions of records
- Sort files of any size with only 2-3 megabytes of RAM. Use more memory to increase the sort performance
- Flexible and easy-to-use scripting language to meet even the most complex sort requirements
- Supports four basic file types: fixed, delimited, counted and explicit
- Over 20 different key types, including textual numbers, date/time (with user-defined masks), fixed decimal, floating point
- Use up to 64 keys in one Sort
- Merge several input files into one output file
- Define filters to remove duplicate records
- Use ranges to limit the number of output records to create ranking lists
- Handle file headers and trailers comfortable with many options
- Make use of all system processors on your Windows NT SMP machines to increase sort speed
- Use up to 64 disks to maximize system throughput
- Run Sorts in batch over night using the Windows 95 / Windows NT scheduling service
- Complete Online-Manual with about 100 pages, packed with samples, reference information and tips
- Extent the library with your own key types for maximum flexibility
- Easy to integrate into all applications written with a 32-Bit language like Visual Basic, Visual C++, Delphi, C++-Builder and others.

How Sort Solution works

The Sort Solution package contains a command line utility called SORTSOL.EXE. With this stand-alone utility you are able to sort files of any size (depending on the Operating System used, see limits) with even the most complex sort criteria.

SORTSOL.EXE requires as input a text file containing statements from the Sort Solution Scripting Language. This file is called a *profile* and contains the name of the input file, the name of the output file, a description of the file format of the input file and a list of KEY statements describing how to sort the input file.
There are additional statements that can be used to optimize the performance of Sort Solution for very big files, define disk and memory usage parameters or create *Filters* and *Ranges* that allow you to restrict the output to records that meet special criteria.

When SORTSOL.EXE is started, it loads the SORTSOL.DLL. This file contains all the Sort Solution logic and is the same file that you will use if you're going to incorporate Sort Solution into your own applications.

After SORTSOL.EXE has completed it's startup, it reads in the profile which was supplied as a command line argument and performs a syntax and logical check on the statements in the profile. If the statements in the profile are syntactical and logically correct, the sort begins.

Depending on the size of the input file and the amount of memory available, Sort Solution uses one or two steps - called *Phases* - to sort the input file into the output file.

Phase I : Sort

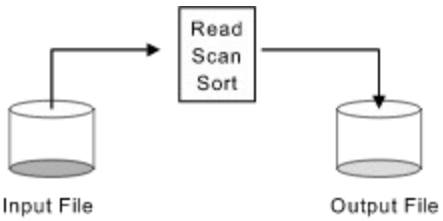
Phase I reads in the input file and extracts the individual records, depending on the file format description in the

profile.

While reading in the file, the records are sorted in parallel and written to the output file. If the input file is to big to fit into memory an once, the sorted records are not written directly to the output file. Instead one or more *temporary work files* (*Merge files*) are created and filled during Phase I.

Each of these temporary files holds a number of presorted blocks - called *Runs* - which are used to create the output file during Phase II.

If the input file is small enough or there is enough memory available to hold the complete file in memory, Sort Solution sorts the file in one single step without creating temporary files.

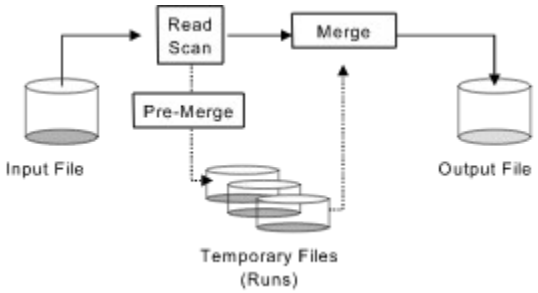


One-Phase Sort

Phase II: Merge

The second phase takes the temporary files created in Phase I and merges the pre-sorted blocks together into one single output file.

This process allows Sort Solution to sort files of nearly any size with only several megabytes of memory.



Two-Phase Sort with Merge

Further, due to the internal technology, this process allows Sort Solution to utilize the available system resources to the maximal extent possible. The reading of the input file, the sorting and the writing to the temporary files all happens in parallel, at the same time.
This allows for a optimal overlap between all required steps in the sort and hence results in a improved overall performance.

Stability

Sort Solution uses a new **stable sort algorithm** that combines a high sort performance with the stability of slower sort techniques.

Stability means, that the relative sequence of records with the same key fields remains unchanged in the output file. Say you have 10 records in the input file with identical keys. The sort identifies these records as »equal« and writes these records in the same sequence to the output file that they have in the input file.

Supported File Types

See also: FILETYPE Keys

Sort Solution implements four different file types. Each of these basic file *formats* can be customized with many options to match exactly the format of the files that you have to sort.

Format FIXED

This file format is used for files with a fixed record length. Every record in the file must have exactly the same length (except the file header). An explicit *record delimiter* is not required. If the file contains record delimiters, they are treated as if they were a natural part of the record itself.

The fields within the records do not need a special *field separator*. Sort Solution adresses individual fields over their *distance in bytes from the beginning of the record* (*Offset*).

This is an extract from a file with a fixed record length. Each line contains exactly one record, the »|« character has been inserted to make the dump more readable, it is not part of the file.

```
100737|05/97|Toothbrush                |000235|  18,30
101738|05/97|Plate, white               |003456|   3,49
101745|05/97|Plate, with decor          |001748|   2,79
104001|06/97|Mirror, antique            |000034| 123,45
```

Every record in this file has a length of 51 byte:

Field 1: 6 Byte, numeric

Field 2: 5 Byte, Date, with the format MM/YY

Field 3: 27 Byte, String

Field 4: 6 Byte, numeric

Field 5: 7 Byte, numeric, floating point with decimal *comma*

The FILETYPE statement for this file in your profile would read

```
FILETYPE(FIXED,51)
```

Format DELIMITED

The file format DELIMITED is used for files with variable record length. Each record has to be delimited with a *record delimiter*. A typical example for this kind of file are text files, comma-separated files, or files that come from a UNIX or Mainframe environment.

Fields within the records must be separated by a special character (*separator*) which can be recognized by Sort Solution while the record is scanned. Sort Solution extracts individual fields from the records by this separator character.

```
100737;05/97;Toothbrush;000235;18,30t
101738;05/97;Plate, white;003456;3,49t
101745;05/97;Plate, white with decor;001748;2,79t
104001;06/97;Mirror, antique;000034;123,45t
```

This is the same file as above, but this time in a delimited format. The delimiter at the end of each record is shown as »t«. In a real file, this can be any character or a combination of characters, e.g. a Carriage Return / Linefeed.

Another example for files in the format DELIMITED are simple text files:

Toothbrush_t
Mirror_t
Towel_t
Coffee_t
Milk_t
Sugar

Pizza¶

Each line (aka *record*) is delimited with a Carriage Return / Linefeed. If the file was created under UNIX, the delimiter usually is only a single Linefeed character.

Format COUNTED

This file format is used to sort files which contain variable length records without an explicit record delimiter. Sort Solution expects files in this format to have *exactly the same number of fields in each record*.

Note Virtually each Database System uses a file format like this for input or *bulk load operations*. If you sort these files with Sort Solution in advance, you can usually expect a performance gain during the load operation because the Database System can load all records in the correct sequence.

Following is a dump from a file in the format COUNTED. Each record consists of 5 fields, separated with semicolons (»;«). Sort Solution can extract the fields during the sort using this separator and also is able to distinguish individual records based on the number of fields that each record must have:

```
100737;05/97;Toothbrush;000235;18,30;101738;05/97;Plate, white
;003456;3,49;101745;05/97;Plate, white with decor;001748;2,79;104001;06/97;Mirror,
antique;000034;123,45;
```

Format EXPLICIT

This format works with files which do contain an *explicit* record length specifier:

```
036100737;05/97;Toothbrush;000235;18,30
037101738;05/97;Plate, white;003456;3,49
048101745;05/97;Plate, white with decor;001748;2,79
042104001;06/97;Mirror, antique;000034;123,45
```

Each record in the file has a leading length specifier, consisting of three digits with the record length in bytes. The fields in each record are separated by a special character (»;«), an explicit record delimiter *is not required*. Sort Solution extracts the records from the file based on the record length given in the first *n* bytes.

The Format FIXED can also handle records which contain a binary length specifier at the beginning of the record. The binary specifier must consist of exactly two bytes (1 Word) in INTEL-Format.

Tip This feature is very useful when you create the input file for the sort from within your own application. In this case, you can use this economic format to save space on your hard disk. Sort Solution can also handle this format very quickly because it needs not to scan the input file for record delimiters. The length of each record is given at the start of each record, which makes it very fast to parse the input file.

The Sort Solution file format that you specify in your profile must match the format of the input file exactly. Due to performance reasons, Sort Solution performs only minimal checks during runtime to check the file format for correctness. If you specify the wrong file format for your input file, the best case will be an unusable output file.

A Special Case for the File Format DELIMITED

Per definition, delimiters are used to *separate* records in a file. A file with DELIMITED format therefore might look like this:

```
Smith¶
Miller¶
Ju¶
Ellison¶
Gates¶
McNealy¶
O'Brian¶
O'Brack¶
Lo
```

The last record has no trailing delimiter, because it has no follow-up records and therefore there is no delimiter

required.

Sort Solution treats this as a special case, for performance reasons. If the last record in a DELIMITED file has no delimiter, Sort Solution automatically appends a delimiter (based on the delimiter definition in the FILETYPE statement). This trick allows Sort Solution to sort the file without a special handling for delimiters and makes the sort a lot faster.
After the sort has finished, Sort Solution automatically removes the delimiter after the last record if it has previously inserted one.

When you use Sort Solution to *concatenate* several input files into one common output file, input files without a delimiter after the last record might lead to an incorrect file format for the resulting file. More on this subject can be found under the topic SORTSOL.EXE.

Records with empty fields

Files in the format DELIMITED, COUNTED and EXPLICIT can contain *empty fields*:

```
100737;05/97;Toothbrush;000235;18,30¶
101738;05/97;Plate, white;003456;3,49¶
101745;05/97;;001748;2,79¶
104001;06/97;Mirror, antique;000034;123,45¶
104004;;No date record;001748;2,79¶
```

An empty field is a field with two consecutive separators (see the two records above that are **bold**-faced). The first bold record has no text, the second has an empty date field.

If such an empty field is part of your sort criteria, Sort Solution treats it as the *smallest possible value* (depending on the data type of the field) and hence sorts the record to the top (or bottom, if you sort in DESCending order) of the file.

Further topics: Keys The Sort Solution Script Language

Keys

See also: Key Types, Supported File Types The Sort Solution Script Language

Sort Solution supports over 20 different key types (aka *data types*) for the definition of sort criteria. This allows to handle almost any sort requirement, even for very complex sort sequences.

You can use up to 64 keys in one sort simultaneously and each key can have different properties.

Each key type supports a *common set of properties*. Depending on the type of the key, extended properties are available which modify the behavior of the key or define additional settings.

Common Key Properties

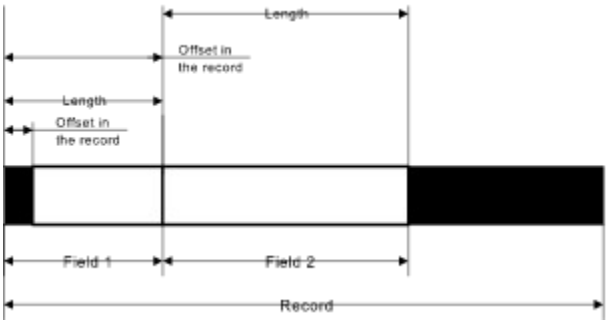
The following properties are common to all key types:

Type	This property defines the data type of the key
Order	The *Order* property defines whether the key sorts ascending or descending. Use the value *ASC* for ascending sort order and *DESC* for descending sort order
Offset	With the *Offset* property you define the start of the key in the record, in bytes from the beginning of the record
Position	File formats with variable key length use the property *Position* instead of *Offset* to define the position of the key within the record. *Position* defines the *field* within the record which should be used as a sort key. Fields are numbered from *1,2,..,n*.
Length	*Length* of the key in byte. For *binary formats*, the length of the key is automatically determined based on the key type.

When you define the position of a key within a record, two different positioning variants are supported: *Absolute Positioning* and *Relative Positioning*.

Absolute Positioning

Absolute Positioning used the common properties *Offset* and *Length* to define the position of a key within a record. This kind of positioning is used for files with a fixed record length or files that use a format without explicit field delimiters (FIXED, EXPLICIT).



Absolute Positioning

Offset is the distance from the beginning of the record in bytes, *Length* is the length of the key (white) in bytes.

For example, take the following records:

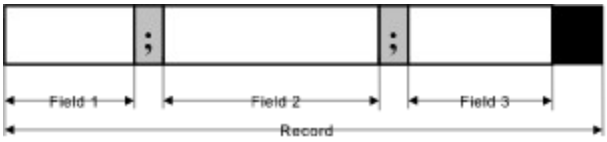
```
012345678901234567890123456789012345
008971Mainboard     250.00
007601Soundcard/LE  065.00
000182Multi-I/O     024.00
```

When you want to create a key which sorts the records on the *Description* field (Offset 6, Length 14), you use

exactly these values: Position = 6, Length = 14. Since files with a fixed record length (Format FIXED) don't have field delimiters, the only way to position a key within the records is to use the actual byte position of the desired field within the record.

Relative Positioning

This looks somewhat different for files with a variable length file format (DELIMITED, COUNTED). Here each record consists of a number of fields, where each field is separated with a special character, the *separator*.



Relative Positioning

When you sort a file with the format DELIMITED or COUNTED, you define the keys *relative to a field number*, starting at 1. Sort Solution determines the length of the field automatically by parsing the complete record.
If a field is empty (to consecutive separators), Sort Solution treats the field as the »smallest« possible value depending on the key type used and sorts the record to the top (or bottom if you use the a descending sort order).

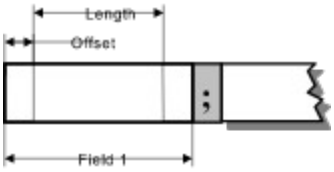
If Sort Solution compares to fields with different lengths, it uses the length of the shorter field to determine the correct sequence if both keys are equal on the length of the shorter field.
If, for example, the fields »Main« and "Mainboard« are compared, the record with the key »Main« is determined as the »smaller« record, according to the sort order used.

```
-- 1 -;---- 2 --;-- 3 -
008971;Mainboard;250.00t
007601;Soundcard/LE;065.00t
008971;Main;250.00t
000182;Multi-I/O;024.00t
```

Each record contains three fields, numbered from 1 to 3. To define a key on the second field, you use the *Position* property of the key and set it to 2. To sort after the first field, you would use 1 for the *Position* property respectively.

Usually, when you use Relative Positioning, the key starts at the beginning of the field and also has the same length as the field.

You can change this standard behavior and use the key properties *Offset* and *Length* to position the key *relative to a field*:



Positioning a key (white) within a field using Relative Positioning

This allows you to use only a *part of a field* as the sort key, which is especially useful when you sort files that have a variable record and field length, but you want use only a specific subsequence of a field as the sort criteria.

Sorting variable length records without field separators

When you have a file with variable length records (e.g. a text file) but without explicit field separators, you can simply take the whole records as *one field*, and position the key within this field using the *Offset* and *Length* properties of the key:

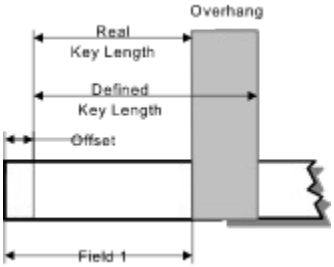
```
ABA05/97Project completedt
DOE09/97Project canceled due to technical reasonst
ABA02/96Consolidation and Testt
COI03/97Regressiont
```

This file has a format with a variable record length and (visually) three fields. Unfortunately, there is no explicit field separator in the file which allows Sort Solution to recognize and separate these three fields.

In this case, you can treat the whole record as one field and use *Offset* and *Length* to specify the key position.

When you use an Offset of 8 and a Length of 0 (Zero), Sort Solution will sort this file after the Description (**bold-faced**).

When you specify a key length of 0, Sort Solution will automatically take the *complete field* (in this case, the complete remainder of the record) as the sort key. So even if you don't know the exact length of the field, Sort Solution will handle this correctly.



Key length for variable length fields

When you only want to sort the first *n characters* of a field, specify the appropriate value in the *Length* property of the key. If the specified length is bigger than the actual length of the field, Sort Solution will automatically cut-off at the end of the field.

Sort Order

The *sort order* of a key defines the sequence in which the records are sorted. When you use an ascending sort order (ASC), the »smallest« key is sorted on top. When you use an descending order (DESC), the »greatest« key is sorted on top. What the »smallest« respective the »greatest« key is depends on the actual key definition.

The sort order is defined on a *per-key basis*, which means that you can sort after one key in ascending order and after another key in descending order simultaneously:

```
06/97;Smith;400t
03/96;Miller;1200t
11/96;Corn;900t
08/97;Miller;700t
08/97;Muller;500t
02/95;Smith;700t
...
```

For a file like this, you probably want to sort each record first after the *Salary* field in *descending* order to put the person with the lowest salary on top. This is the *primary sort key*.

Within each salary group (persons with the same salary), you want to sort the persons by name in the usual ascending order:

```
06/97;Smith;400t
08/97;Muller;500t
02/95;Smith;700t
03/96;Miller;700t
11/96;Corn;900t
08/97;Miller;1200t
...
```

Stability

Sort Solution uses a new **stable sort algorithm** that combines a high sort performance with the stability of slower sort techniques.

Stability means, that the relative sequence of records with the same key fields remains unchanged in the output file. Say you have 10 records in the input file with identical keys. The sort identifies these records as »equal« and writes these records in the same sequence to the output file that they have in the input file.

Further Information: Key Types

Key Types

Sort Solution supports a variety of different key types for nearly every data type and file format.

Generic Keys	For binary comparison without any specific data type or interpretation
Strings	Character arrays, Strings in ANSI/ASCII format, Strings with special characters and National Language Strings
Binary Keys	Binary keys in INTEL or IEEE format (Integer, Unsigned Integer, Long Integer, Float, Double...)
Numbers as Strings	Strings containing numbers, e.g. "1000" or "123.45"
Fixed Decimal	Fixed decimal format (e.g. COBOL)
Date	Date, with masks
Time	Time, with masks
User-defined Sequences	User- defined character sequences
User Keys	Special or customized key types in external DLL's

KEY: Generic Keys

Main Topic: Keys See also: Key Types, The Sort Solution Script Language

Generic

Length: Variable
Type: Binary
Range: -
Syntax: KEY(Generic,[ASC|DESC],<Position>,<Offset>,<Length>)

The key type *Generic* sorts any kind of data without a special interpretation. It performs a byte-by-byte binary comparison of the keys.

The key type Generic is the fastest key type available because it performs absolutely no interpretation of the field content. When you need to achieve the fastest possible sort speed, you should use Generic whenever possible.

```
KEY(Generic,ASC,0,2,12)
```

This key statement sorts the input file (FIXED format, see Supported File Types) based on a key that starts at offset 2 of the record on a length of 12 bytes. The records are sorted in ASCending order:

0123454
0123450
0123440

After the sort, you will have:

0123440
0123450
0123454

When you want to sort in descending order, exchange the ASC specifier in the key statement with DESC:

```
KEY(Generic,DESC,0,2,12)
```

In this case, after the sort the file will look like this:

0123454
0123450
0123440

If two keys with different lengths are compared, the shorter key wins when both keys are equal within the shorter length:

Mill
Miller

KEY: Strings

Main Topic: Keys See also: Key Types, The Sort Solution Script Language

String

Length: Variable
Type: Binary
Range: -
Syntax: KEY(String,[ASC|DESC],<Position>,<Offset>,<Length>)

The key type *String* compares records based on the ASCII code of their characters. *String* is a synonym for KEY: Generic Keys.

StringN

Length: Variable
Type: Binary
Range: -
Syntax: KEY(StringN,[ASC|DESC],<Position>,<Offset>,<Length>)

The key type *StringN* sorts records based on the ASCII value of their characters. *StringN* does a *case-insensitive* comparison, so »MILLER«, »Miller«, and »miller« are treated as equal.

String and *StringN* perform a comparison based on the ASCII codes of the characters in the record. If your records contain special *locale characters*, like the German Umlauts (»Ä«, »Ö«, »Ü«) or special French or Latin characters, the resulting sort sequence will be wrong.
This is because the ASCII code (or ANSI code under Windows) was created in the early days of »Personal Computing« and hence contains virtually *no support for language or country specific characters*.

National Languages Support (NLS)

An outstanding feature of Sort Solution is the support for *National Language keys*. NLS keys are keys that are aware of country specific character sets (code pages) and also do adhere to specific sort requirements for different countries.

For example, the German sort order for telephone books imposes a special sequence for the German Umlauts. Most of the applications written specific to support strings containing the special German characters use this sequence when it comes to sorting.

```
Müller↵
März↵
Muller↵
Muellerr↵
Älter↵
Alter↵
```

When you sort this file using one of the key types *String* or *Generic (or the DOS SORT command)*, the result will look like this:

```
Älter↵↵
Alter↵↵
Müller↵↵
März↵↵
Muellerr↵↵
Muller↵↵
```

This is wrong, at least for a German viewer. The correct sort order should be:

```
Älter↵↵
Alter↵↵
```

Märzt
Müllert
Muellert
Mullert

The character »ä« has to be treated as an »ae« when it is sorted. Therefore »ä« (ae) comes in front of »ü« (ue) in a sorted sequence.

The French language has even more complex sort order requirements, and the same applies to most other European languages.

Language Sensitive Keys

Sort Solution supports the two key types *StringNLS* and *StringNLSN* which can handle country specific sort rules and character sets correctly.

StringNLS

Length: Variable
Type: Binary
Range: -
Syntax: KEY(StringNLS,[ASC|DESC],<Position>,<Offset>,<Length>,<Lang>,<Sublang>)

StringNLSN

Length: Variable
Type: Binary
Range: -
Syntax: KEY(StringNLS,[ASC|DESC],<Position>,<Offset>,<Length>,<Lang>,<Sublang>)

Both key types take the same parameters as String or StringN respectively. StringNLSN is the case-insensitive alternative to *StringNLS* and performs comparisons without differentiating between upper- and lower-case.

Language Identifiers

Both keys take two additional arguments: a *primary language identifier* and a *secondary language identifier*. Both identifiers together built a *language descriptor* which imposes a specific sort order based on country specific sort rules.

The *primary language identifier* identifies the general language, for example LANG_GERMAN for the German language. The secondary language identifier is used to specify a country-specific sub-language, like SUBLANG_GERMAN_SWISS for Switzerland or SUBLANG_GERMAN_LUXEMBOURG for Luxembourg.

The French language (LANG_FRENCH) has for example 5 different sub-languages:

SUBLANG_FRENCH
SUBLANG_FRENCH_BELGIAN
SUBLANG_FRENCH_CANADIAN
SUBLANG_FRENCH_SWISS
SUBLANG_FRENCH_LUXEMBOURG

To sort a file containing language specific characters, use one of the key types *StringNLS* or *StringNLSN* with the appropriate language identifier:

```
KEY(StringNLS, ASC, 1, 0, 0, LANG_GERMAN, SUBLANG_GERMAN)
```

or

```
KEY(StringNLS, ASC, 1, 0, 0, LANG_FRENCH, SUBLANG_FRENCH_CANADIAN)
```

List of Supported Language Identifiers

The following section lists all primary and secondary languages supported by Sort Solution.

Please note that Sort Solution performs no checks to make sure that the combination of primary and secondary language you choose is valid.
If you combine two unrelated language identifiers, the resulting sequence is undefined!

Primary Languages

LANG_NEUTRAL
LANG_AFRIKAANS
LANG_ALBANIAN
LANG_ARABIC
LANG_BASQUE
LANG_BELARUSIAN
LANG_BULGARIANLANG_CATALAN
LANG_CHINESE
LANG_CROATIAN
LANG_CZECH
LANG_DANISH
LANG_DUTCH
LANG_ENGLISH
LANG_ESTONIAN
LANG_FAEROESE
LANG_FARSI
LANG_FINNISH
LANG_FRENCH
LANG_GERMAN
LANG_GREEK
LANG_HEBREW
LANG_HUNGARIAN
LANG_ICELANDIC
LANG_INDONESIAN
LANG_ITALIAN
LANG_JAPANESE
LANG_KOREAN
LANG_LATVIAN
LANG_LITHUANIAN
LANG_NORWEGIAN
LANG_POLISH
LANG_PORTUGUESE
LANG_ROMANIAN
LANG_RUSSIAN
LANG_SERBIAN
LANG_SLOVAK
LANG_SLOVENIAN
LANG_SPANISH
LANG_SWEDISH
LANG_THAI
LANG_TURKISH
LANG_UKRAINIAN
LANG_VIETNAMESE

Secondary Languages

SUBLANG_NEUTRAL
SUBLANG_DEFAULT
SUBLANG_SYS_DEFAULT
SUBLANG_ARABIC_SAUDI_ARABIA
SUBLANG_ARABIC_IRAQ
SUBLANG_ARABIC_EGYPT
SUBLANG_ARABIC_LIBYA
SUBLANG_ARABIC_ALGERIA
SUBLANG_ARABIC_MOROCCO
SUBLANG_ARABIC_TUNISIA
SUBLANG_ARABIC_OMAN
SUBLANG_ARABIC_YEMEN
SUBLANG_ARABIC_SYRIA
SUBLANG_ARABIC_JORDAN
SUBLANG_ARABIC_LEBANON
SUBLANG_ARABIC_KUWAIT
SUBLANG_ARABIC_UAE
SUBLANG_ARABIC_BAHRAIN
SUBLANG_ARABIC_QATAR
SUBLANG_CHINESE_TRADITIONAL
SUBLANG_CHINESE_SIMPLIFIED
SUBLANG_CHINESE_HONGKONG
SUBLANG_CHINESE_SINGAPORE
SUBLANG_DUTCH
SUBLANG_DUTCH_BELGIAN
SUBLANG_ENGLISH_US
SUBLANG_ENGLISH_UK
SUBLANG_ENGLISH_AUS
SUBLANG_ENGLISH_CAN
SUBLANG_ENGLISH_NZ
SUBLANG_ENGLISH_EIRE
SUBLANG_ENGLISH_SOUTH_AFRICA
SUBLANG_ENGLISH_JAMAICA
SUBLANG_ENGLISH_CARIBBEAN
SUBLANG_ENGLISH_BELIZE
SUBLANG_ENGLISH_TRINIDAD
SUBLANG_FRENCH
SUBLANG_FRENCH_BELGIAN
SUBLANG_FRENCH_CANADIAN
SUBLANG_FRENCH_SWISS
SUBLANG_FRENCH_LUXEMBOURG
SUBLANG_GERMAN
SUBLANG_GERMAN_SWISS
SUBLANG_GERMAN_AUSTRIAN
SUBLANG_GERMAN_LUXEMBOURG
SUBLANG_GERMAN_LIECHTENSTEIN
SUBLANG_GERMAN_PHONE_BOOK
SUBLANG_ITALIAN
SUBLANG_ITALIAN_SWISS
SUBLANG_KOREAN
SUBLANG_KOREAN_JOHAB
SUBLANG_NORWEGIAN_BOKMAL
SUBLANG_NORWEGIAN_NYNORSK
SUBLANG_PORTUGUESE
SUBLANG_PORTUGUESE_BRAZILIAN
SUBLANG_SERBIAN_LATIN
SUBLANG_SERBIAN_CYRILLIC
SUBLANG_SPANISH
SUBLANG_SPANISH_MEXICAN
SUBLANG_SPANISH_MODERN
SUBLANG_SPANISH_GUATEMALA
SUBLANG_SPANISH_COSTA_RICA
SUBLANG_SPANISH_PANAMA
SUBLANG_SPANISH_VENEZUELA
SUBLANG_SPANISH_COLOMBIA
SUBLANG_SPANISH_PERU
SUBLANG_SPANISH_ARGENTINA
SUBLANG_SPANISH_ECUADOR
SUBLANG_SPANISH_CHILE
SUBLANG_SPANISH_URUGUAY
SUBLANG_SPANISH_PARAGUAY
SUBLANG_SPANISH_BOLIVIA
SUBLANG_SPANISH_EL_SALVADOR
SUBLANG_SPANISH_HONDURAS

SUBLANG_SPANISH_NICARAGUA
SUBLANG_SPANISH_PUERTO_RICO
SUBLANG_SWEDISH
SUBLANG_SWEDISH_FINLAND
SUBLANG_SPANISH_DOMINICAN_REPUBLIC

KEY: Binary Keys

Main Topic: Keys See also: Key Types, The Sort Solution Script Language

Sort Solution has a built in support for the most frequently used binary data types on Windows 95 and Windows NT. These formats are especially useful for programmers when they want to sort files that are created from within an application. Instead of converting a file in a »readable« text format for sorting, they can use the binary data types supplied by Sort Solution to sort the files directly.

ShortInt

Length: 2 Byte
Type: INTEL, 2 Byte signed Integer
Range: -32768 to 32767
Syntax: KEY(ShortInt,[ASC|DESC],<Position>,<Offset>)

UShortInt

Length: 2 Byte
Type: INTEL, 2 Byte unsigned Integer
Range: 0 to 65535
Syntax: KEY(UShortInt,[ASC|DESC],<Position>,<Offset>)

LongInt

Length: 4 Byte
Type: INTEL, 4 Byte signed Integer
Range: -2.147.483.648 to 2.147.483.647
Syntax: KEY(LongInt,[ASC|DESC],<Position>,<Offset>)

ULongInt

Length: 4 Byte
Type: INTEL, 4 Byte unsigned Integer
Range: 0 to 4.294.967.295
Syntax: KEY(ULongInt,[ASC|DESC],<Position>,<Offset>)

LongInt64

Length: 8 Byte
Type: INTEL, 8 Byte signed Integer
Range: -9.223.372.036.854.775.808 to 9.223.372.036.854.775.807
Syntax: KEY(LongInt64,[ASC|DESC],<Position>,<Offset>)

ULongInt64

Length: 8 Byte
Type: INTEL, 8 Byte unsigned Integer
Range: 0 to 18.446.744.073.709.551.614
Syntax: KEY(ULongInt64,[ASC|DESC],<Position>,<Offset>)

Float

Length: 4 Byte
Type: IEEE, 4 Byte Floating Point
Range: 3.4E +/- 38 (7 digits)
Syntax: KEY(Float,[ASC|DESC],<Position>,<Offset>)

Double

Length: 8 Byte
Type: IEEE, 8 Byte Floating Point
Range: 1.7E +/- 308 (15 digits)
Syntax: KEY(Double,[ASC|DESC],<Position>,<Offset>)

LDouble

Length: 10 Byte
Type: IEEE, 10 Byte Floating Point
Range: 1.2E +/- 4932 (19 digits)
Syntax: KEY(LDouble,[ASC|DESC],<Position>,<Offset>)

The data type *LDouble* is currently mapped to *Double*, so the two key types a *exchangeable*.

KEY: Numbers as Strings

Main Topic: Keys See also: Key Types, The Sort Solution Script Language

Many files that are created/exported from applications do contain fields with numbers in form of strings:

```
ABA;10089;Mainboard;123.45
AHEJ;19283;Speaker;456.34
LUHZ;192823;Mouse, MS;45.50
...
```

When you want to sort a file after these fields, you cannot simply use the Generic or String key type. If you use one of these simple key types, the resulting sort order will be wrong in most cases:

```
10¶
12¶
11¶
9¶
7¶
1¶
2¶
6¶
```

When you sort a file after these fields, using a *Generic* or *String* key type, you will get this sequence:

```
1¶
10¶
11¶
12¶
2¶
6¶
7¶
9¶
```

This is because how the *Generic* and *String* keys work. They compare two fields character per character from the beginning of the field until they find a difference or the end of one or both fields is reached.

Take for example the two fields »12« and »2«. You might say that »2« is smaller than »12« and from a numerical point of view you're completely right. But when the comparison is performed, the first characters compared are the »1« from »12« and the »2« from the second key:

12¶
2¶

Clearly, the first character of »12« is smaller than the »2« and hence the »12« is treated as »smaller«.

> This is not a unique behavior for Sort Solution. You can see this in nearly any application that sorts records or strings that contain numbers.
> Give it a try: Create several folders on your hard disk and name them 1,2,...10,11,12,...
> Then look at the folders in the Windows Explorer or use the DIR command on the command line with the /O option. You will get the same »wrong« sort order here.

The problem is, that the fields are not treated as numerical values, but treated as string literals.

To solve this problem, Sort Solution contains several key types that are able to sort fields containing numbers after their numerical value.

IntS

Length: Variable
Type: Text
Range: Sign plus 10 digits maximum
-2147483648 to 2147483647
Syntax: KEY(IntS,[ASC|DESC],<Position>,<Offset>,<Length>)

The key type *IntS* interprets the bytes of the key as a string containing a number and transforms this number into the corresponding numerical value before a compare operation is performed. The field sorted with this key type must contain only digits in the range »0«..»9« and optionally a sign (»-« or »+«). The maximum number of digits allowed is 10.

Note If the key contains an invalid number or a character that is not allowed for this key type, Sort Solution automatically uses the value 0 for the key and issues *no warning*.

If you sort the records from above using the key type IntS, you will get the right sort order:

```
KEY(IntS,ASC,1,0,0)

1t
2t
6t
7t
9t
10t
11t
12t
```

UIntS

Length: Variable
Type: Text
Range: Maximum 10 digits
0 to 4294967295
Syntax: KEY(UIntS,[ASC|DESC],<Position>,<Offset>,<Length>)

This key type interprets the digits in the sorted field as an unsigned integer with a maximum of 10 digits. If the key does contain any invalid characters, it's value is set to 0 without a warning.

IntS64

Length: Variable
Type: Text
Range: Sign plus maximum 19 digits
–9223372036854775808 to 9223372036854775807
Syntax: KEY(IntS64,[ASC|DESC],<Position>,<Offset>,<Length>)

For extremely large values (e.g. monetary values in cents) you can use this key type. Except the increased range it works exactly as *IntS*.

UIntS64

Length: Variable
Type: Text
Range: Maximum 19 digits
0 to 18446744073709551614
Syntax: KEY(UIntS64,[ASC|DESC],<Position>,<Offset>,<Length>)

Like *UIntS*, but with an increased range of 64 bits.

DoubleS

Length: Variable

Type: Text
Range: 1.7E +/- 308 (15 digits)
Syntax: KEY(DoubleS,[ASC|DESC],<Position>,<Offset>,<Length>,<Decimal Point Symbol>)

For floating point values (numbers with a decimal point), Sort Solution provides the key type *DoubleS*. This key type interprets the contents of the field as an numerical value with a sign, a decimal comma and optionally an exponent and mantissa.

Allowed characters for this key type are:

```
<Blank><Sign><Digit(s)>.<Digit(s)>
```

optionally followed by an exponent

```
[e|E]<Sign><Digit(s)>
```

Examples:

```
123.50
18
10e10
45,03       ## special case, see below!
29.5
```

The Decimal Point

Depending on the origin of the file you are sorting, the decimal point is actually a point (».«) or a comma (»,«). Germany for example uses a decimal comma instead a decimal point:

123.45	U.S.
100,000.99	US
123,45	Germany
100.000,99	Germany

To be able to handle all kinds of floating point numbers, Sort Solution allows you to specify which character should be interpreted as the decimal comma in floating point numbers. You define the desired character as the last argument in the KEY statement:

```
; For the U.S.
KEY(DoubleS,2,0,0,".")

; For Germany
KEY(DoubleS,2,0,0,",")
```

KEY: Fixed Decimal

Main Topic: Keys See also: Key Types, The Sort Solution Script Language

FixDecimal

Length: Variable
Type: Text
Range: 1.7E +/- 308 (15 digits)
Syntax: KEY(FixDecimal,[ASC|DESC],<Position>,<Offset>,<Length>,<Decimal Places>)

Some Computer Systems, e.g. Mainframes, use a fixed decimal format when floating point numbers are written to files. COBOL programs for example often use a fixed decimal notation in files that contain floating point numbers.

A fixed decimal number contains *no* explicit decimal point, instead it uses a predefined number of digits for the part after the decimal place:

```
1235500;t      ð 123,5500 (4 digits after the decimal place)
7859;t         ð 78,59    (2 digits after the decimal place)
```

Sort Solution offers the *FixDecimal* key type for these kind of numbers. You specify the number of digits that are used as an parameter to the KEY statement:

; A Fixed Decimal key with four digits

KEY(FixDecimal,1,0,0,4)

KEY: Date

Main Topic: Keys See also: Key Types, KEY: Time, The Sort Solution Script Language

Date

Length: Variable, depends on *Mask*
Type: Text
Range: -
Syntax: KEY(Date,[ASC|DESC],<Position>,<Offset>,"<Mask>",{<Limit>})

Many files do contain *date fields* in form of strings, e.g.

```
12.02.1997 (DayDay.MonthMonth.YearYearYearYear)
12/02/1997 (DD.MM.YYYY)
12/02/97   (MM.DD.YY)
02/97      (MM.YY)
01-01-1997 (DD-MM-YYYY)
1997/02/01 (YYYY/MM/DD)
...
```

Depending on the format used for the date and the country format in which the date is stored, it is impossible to sort a file after these fields. Only the last date field from the fields listed above is suitable for sorting because it uses the format YYYY/02/01, where the year is in front of the month which is in front of the day. If you use this format for dates in files, you can sort the file on these fields using the Generic key type. In all other cases, you should use the key type *Date* instead.

Date uses a *mask* to specify the format of the date. This allows you to sort virtually any date format you can think of. For example, to sort a date in the format

MM/DD/YYYY

you will use a Date key like this in your profile:

```
KEY(Date,ASC,1,0,"MM/DD/YYYY")
```

With the mask supplied, Sort Solution is able to convert the textual representation of the date in the field into a numeric format (a so-called *Julian Date*) and use this numeric representation to perform comparisons.

The Mask

The mask must match exactly the format of your date string. The mask uses the special characters D, M, and Y to specify the date content of your field:

D	Day	D or DD
M	Month	M or MM
Y	Year	YY or YYY

The combination of these three tokens builds up the date format description used by Sort Solution. Any other character in the mask serves only as a placeholder and must match exactly the actual content of the date fields sorted.

Mask	Example	Description
"MM/DD/YY"	01/01/97 1/2/97 3/10/97 11/23/88	A date in western standard format, with a two-digit year. Sort Solution is able to handle even missing digits (only one digit for Month or Day)
"DD.MM.YYYY"	01.01.1999 31.12.2003 19.12.98	German date format with a four-digit year
"MM.YY"	05.92 5.92	A date format with only a month and year. Sort Solution treats the missing day value as the first day in the given month

"MM/YYYY"	05/1992 2/2003 7/1997	Like above, but with a four-digit year
"MM-DD-YY"	08-01-96 8-1-96	A date format with the »-« as the separator sign
"YY,MM,DD"	86,01,26 86,2,7	Unusual date format with a »,« separator

Special Cases

Date is able to handle most common exceptions for date formats:

DD	Handles one- and two-digits days, e.g.: "5/01/1996" or "14/01/1996"
MM	Handles one- and two-digits month, e.g.: "5/1/1996" or "05/12/1996"
YYYY	Handles two- and four-digits years, e.g.: "05/01/1999" or "05/01/99"

Day with values greater than 31 are treated as 31.

Month with values above 12 are treated as 12.

All other incorrect values are treated as 1.

Note When you use a mask that does not match the given date field, the results of the sort are *undefined*.

Attention

Sort Solution is only able to handle these special cases if it able to recognize them. This is only possible if the length of the mask and the length of the date field don't match:

```
Mask : MM/DD/YY
Field: 12/1/97
```

Here we have a case where the mask is longer than the actual field in the record. This allows Sort Solution to handle the missing day digits correctly.

```
Mask : MM/DD/YY
Field: 12/1/97 ;Miller;
```

In this case, the field length is filled up with an extra blank (*denoted by an underline*), so the length of the mask matches the length of the field, although the date field is missing a day digit. In this (unusual and rare) case the matching algorithm of Sort Solution fails.

Year 2000

The year 2000 is a special issue when it comes to sorting, especially when the date fields in the file use only two digits for the representation or years.

Imagine a file that contains date fields with a two-digit year where some of the records have a date after the year 2000:

```
...;01/01/99
...;01/01/97
...;01/01/98
...;01/01/00
```

When you sort a file like this using the *Date* key type, you will get obviously the wrong sequence:

```
...;01/01/00
...;01/01/97
...;01/01/98
...;01/01/99
```

This results from the (20)00 being numerical smaller than all other year values in the file.

The *Date* key type has an optional parameter which allows you to specify a so-called *threshold*. This value is the last argument to the Date KEY statement and specifies which keys should be treated as Year 2000 keys.

For example:

```
KEY(Date,ASC,2,0,"MM/DD/YY",70)
```

This key specifies that all dates that have a year **less than 70** should be treated as 2000+. If Sort Solution uses this key definition and finds a date like *01/01/02* in your file, it sorts this records as if it has a year value of **2002**.

If you face the problem of sorting a file with dates behind the Year 2000, use the date correction and use a value for the threshold which is somewhat smaller than any 19th century key in your file.

KEY: Time

Main Topic: Keys See also: Key Types, KEY: Date, The Sort Solution Script Language

Time

Length: Variable, depending on Mask
Type: Text
Range: -
Syntax: KEY(Time,[ASC|DESC],<Position>,<Offset>,<Mask>)

The *Time* key works similar to the Date key, but for time values. Like *Date* it uses a mask to specify the format of the time field to be sorted:

The Mask

H	Hour	H or HH
M	Minute	M or MM
S	Second	S or SS

The combination of these three tokens builds up the time format description used by Sort Solution. Any other character in the mask serves only as a placeholder and must match exactly the actual content of the time fields sorted.

"HH.MM:SS"	12.00:00 0.42:01	Various German time formats. Like the *Date* key type, the *Time* key type is able to handle the most common special cases, like single-digit hour or single-digit minutes.
"HH:MM.SS"	12:00.00 0:42.00	American/English time format
"HH:MM"	12:00	Only hour and minute. Sort Solution treats the missing seconds as 00
"MM/HH/SS"	59/23/57	Extraordinary time format

Note When you use a mask that does not match the given time field, the results of the sort are *undefined*.

KEY: User-defined Sequences

Main Topic: Keys See also: Key Types, KEY: Date, The Sort Solution Script Language

UDS

Length: Variable
Type: Text
Range: -
Syntax: KEY(UDS,[ASC|DESC],<Position>,<Offset>,<Length>,<Sequenz>)

The key type UDS (User defined Sequence) gives you the maximum control on how Sort Solution sorts. With UDS you are able to give each single character in the ASCII character set a new value which results in a complete new sort sequence.

Normally, when Sort Solution sees fields like this:

```
AB
CD
DA
BC
```

it sorts the fields bases on the ASCII code of the individual characters. The character »A« for example has an ASCII value of 64 (see the ASCII table in the appendix), »B« has 66 and so on.

After the sort, using a String or Generic key, you will get a sequence like this:

```
AB
BC
CD
DA
```

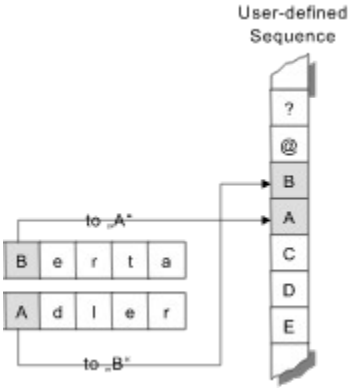
You can change the sort order when you use the DESC specifier in your KEY statement. In this case you will get

```
DA
CD
BC
AB
```

What you can't do with normal keys is to assign a *complete new sequence* to the characters in the ASCII table. Perhaps you want to sort all records starting with a »C« on top of »B« or all fields that start with »WZ« in front of »CE« without interfering with the other key fields in your file.

The key type UDS allows you exactly to do that. UDS keys use a sort order that depends on a *virtual ASCII table* or *Sequence* that is defined by you.

A Sequence contains exactly 256 characters which define the new sort order. If your sequence contains for example the character »B« on position 65 (which is normally allocated by »A«) and a »A« on position 66, Sort Solution will sort »B« in front of »A« if you sort in ascending order.



A user-defined sequence

A user-defined sequence serves as a *look-up table* for Sort Solution. Instead of using the original ASCII code for the characters in the key field, Sort Solution looks up the new value in the sequence defined for the current UDS key.

UDS keys can be used for a variety of purposes. For example, you can create an UDS sequence which sorts the digits »0« to »9« behind the letters »A« to »Z«. This can be very useful under some circumstances.
Sometimes an UDS key allows you to sort a file with a somewhat crude record format when every other key fails to create the correct sequence. Try it out!

KEY: User Keys

Main Topic: Keys See also: Key Types, KEY: Date, The Sort Solution Script Language

User

Length: Variable
Type: -
Range: -
Syntax: KEY(User,[ASC|DESC],<Position>,<Offset>,<Length>,"<DLL Name>","<Function Name>"{,"<Data>"})

<DLL Name>	Name and path to the DLL that contains the user key
<Function Name>	Name of the *Compare function* in the external key DLL. A key DLL can contain any number of Compare functions. With this parameter you specify which function should be used. The name is **case-sensitive**.
"<Data>"	Optional. The content of this string depends on the key DLL and the Compare function used. Please consult the documentation of the key DLL for more details.

One of the most compelling features for Developers it the extendibility of Sort Solution. In addition to the variety of built in key types which already handle even exotic data formats, Sort Solution implements an *Add-In concept* which allows you to create and use *new* key types contained in external DLL's.

In your profile, a KEY statement using a key imported from an external DLL may look like this:

```
KEY(User,ASC,0,17,174,"mykey.dll","Compare")
```

This KEY statement uses the compare function "Compare" from the external key DLL "mykey.dll". (*This is, by the way, the key DLL which is delivered as a source code example on how to write external key DLLs for Sort Solution*).

You can continue to use all the features of Sort Solution when you use keys from external DLL's, but you gain the possibility to add any kind of key type.

Say for example, your file contains records that store GPI (geographical) data and you want to sort it after specific criteria. Since Sort Solution doesn't know anything about this data type, it is probably not able to sort the records correctly.

In this case, you may want to create a key DLL containing a specialized key type which can handle the GPI records. Even if you're not a programmer, you will be very likely be able to find a person who is able and willing to create a key DLL for you.

Then you can use the new key type like any of the built-in key types of Sort Solution in your profiles:

```
KEY(User,ASC,0,0,36,"GPI.dll","GPICompare")
```

For more information on the subject of key DLL's refer to KEY DLL's.

SORTSOL.EXE

Keys, Supported File Types, The Sort Solution Script Language SORTSOL.EXE Return Codes, Error Messages,

SORTSOL.EXE is the command line application that comes with the Sort Solution package. It allows you to sort files of any size (see: Limits) from the Windows 95 or Windows NT command line.

SORTSOL.EXE relies on the power of the underlying Sort Solution Library, which you can also use in your own applications. See Incorperating Sort Solution into Your Applications for more details.

Executing SORTSOL.EXE

SORTSOL.EXE is controlled by Sort Solution Profiles, text files containing statements from the the Sort Solution Script Language. It takes the filename of the profile as a command line argument, executes the statements in the file and returns after the sort has completed.

Syntax

```
sortsol {-?|-h} {-C}<Profile> {<Input file>} {{+}<Output file>} {{+}<Logfile>}
```

Arguments in parentheses are optional and can be omitted. If you use these arguments, you can override settings in the profile. This can be useful if you execute SORTSOL.EXE from within a batch file or with the *Windows Scheduling Service* for unattended sorts.

{-?,-h}	Displays the Online Help. This command assumes that the file SORTSOL.HLP is in the same directory as SORTSOL.EXE or in the Windows\System or \Windows\System32 directory.
<Profile>	The name and path of the profile to be executed. If no path is given, the profile must be located in the current directory
<Input file>	Name of the input file. This parameter is optional. If it is given, it overrides the INPUTFILE statement in the profile
<Output file>	Name of the output file. This parameter is optional. If it is given, it overrides the OUTPUTFILE statement in the profile
<Logfile>	Name of the logfile. This parameter is optional. If it is given, it overrides the LOGFILE statement in the profile
+	Append. If you use a »+« in front of the <Output file> or <Logfile> parameters, the new data is appended to the file. This option is used to *concatenate* several input files to one common output file or to append new records to an existing logfile. See the OUTPUTFILE and LOGFILE statements for more details
-C	Creates a new profile with default settings.

Creating a new profile

To create a new profile from scratch which then can be edited using you favorite editor, use the -C flag and a filename:

```
sortsol -Csort.ssp
```

When you type this statement at the command prompt (DOS box), Sort Solution will create a new profile named *sort.ssp* in the current directory.

SSP stands for *Sort Solution Profile* and is the standard extension for profiles.

Sorting

```
sortsol sort.ssp
```

This command executes SORTSOL.EXE using the settings from *sort.ssp*. The profile must contain INPUTFILE and OUTPUTFILE statements.

```
sortsol sort.ssp input.txt
```

This command executes SORTSOL.EXE using the settings from *sort.ssp*. Since the name of the input file is given at the command line and hence the profile's INPUTFILE statement will be ignored.

```
sortsol sort.ssp input.txt output.txt
```

This command executes SORTSOL.EXE using the settings from *sort.ssp*. Since the names of the input file and the output file are given at the command line, the profile's INPUTFILE and OUTPUTFILE statements will be ignored.

You can also use the same file for input and output. This way Sort Solution will sort the input file »in-place«:

```
sortsol sort.ssp mydata.dat mydata.dat
```

Concatenating Files

From time to time you may want to merge several input files into one common output file.

Say you have four files a.dat, b.dat, c.dat and d.dat and you want to create one single output file containing the sorted content from *all* these input files.

First, you sort each of the input files into the file out.dat. Please note, that you must use the Append option (+) for the second, third and fourth sort to append the new content to the output file:

```
sortsol sort.ssp a.dat out.dat
sortsol sort.ssp b.dat +out.dat
sortsol sort.ssp c.dat +out.dat
sortsol sort.ssp d.dat +out.dat
```

The last step is now to sort the output file in-place:

```
sortsol sort.ssp out.dat out.dat
```

Now you have the data from all input files in one sorted file. Of course, you can use different profiles for each of the sorts. For example, it might be useful to remove duplicate records from the input files using a Filter or to limit the range of records in the output file (cut-off, top-10) with a RANGE statement.

> Please note that Sort Solution does not append any delimiters at the end of the input files when you concatenate files into one output file. When you concatenate DELIMITED files, your input files must already contain a delimiter after the last record or your output file will not have delimiters after some of the records.

Aborting SORTSOL.EXE

You can abort SORTSOL.EXE at any time using <Ctrl>+<Break> or <Ctrl>+<C>. Sort Solution displays a »Acknowledged« message on the screen, cleans up all open files, frees all allocated memory and returns as soon as possible. Due to the inner workings of Sort Solution, it can take some time before Sort Solution returns to the command line.

Controlling the Priority

You can control the amount of processing time that Sort Solution demands from the Operating System with the PRIORITY statement in your profile. This is especially important if you use Sort Solution in batch mode or on a Database Server on the network where you must have full control over the performance of the computer.

Errors and Error Messages

When Sort Solution finds an error in the profile or an error occurs during the sort, an error message is displayed on the screen and SORTSOL.EXE is aborted.

Errors in the Profile

When Sort Solution finds errors in your profile, like wrong or missing statements, typos or some other kind of error,

a diagnostic message with four parts is displayed:

- An Error Code
- The line number where the error was found
- A error message
- The contents of the line in which the error was found

An example:

```
Error 22003 (2): Unknown keyword found
"Thread(4)"
```

Sort Solution has diagnosed an error in line 2 of the profile. The code of the error is 22003, which stands for "Unknown keyword found". The line in which the error was found is displayed directly below the error message. The reason for this error is a misspelled keyword. »Thread« should read »Thread**s**«.

Runtime Errors

This kind of error occurs during the sort. A possible problem for an error could be full disk, a physical read or write failure or a wrong data format in the input file.

Runtime errors are displayed only with an error code and an appropriate message, e.g.:

```
Error: (20100) Source file not found or not accessible
```

A list of all possible error codes can be found under the topic Error Messages.

The Sort Solution Script Language

Command Overview, Keys, Supported File Types, Error Messages

The key to the power of Sort Solution is the Sort Solution *Script Language* (*SSSL*). Using the statements of the language you are able to describe all the information that Sort Solution needs to sort, merge, or filter a file.

Please refer to the following sections for more information about a specific topic:

Profiles	Learn how to create, edit and use profiles
Command Overview	This section gives you an comprehensive overview of all statements supported by the Sort Solution Script Language
Keys	*Keys* are used to define the sort criteria for your files. Each profile must at least contain one KEY statement
Supported File Types	This section describes all supported file types and when and how to use them
Performance Tuning	In this section you get some tips to improve the overall performance of Sort Solution for very large files.

Profiles

The Sort Solution Script Language

A *profile* is a text file containing a sequence of statements from the Sort Solution Script Language.

Every line in a profile contains exact one statement, a comment or it is an empty line. Every line must be delimited with a Carriage Return / Linefeed.

You can use your favorite text editor (e.g. Windows Notepad or Wordpad) to create profiles. Any other editor which creates text files without embedded control characters will also do.

Comments

You can use comments in your profile to describe single statements or write down some information about how to use the profile. A comment starts with a semicolon as the first character in the line and ends at the end of the line. Multi-line comments are not supported.

```
; This is a comment
; This is anothe comment
```

Every line that does not contain a valid statement must be a comment or an empty line.

Statements

A statement is a sequence of characters that built up a Sort Solution command.
Every statement must contain a *keyword* , two parentheses and must adhere the following syntax:

```
Keyword(<Argument 1>,<Argument 2>, ... , <Argument n>)
```

Each statement must be on a single line. Line breaks in statements are not allowed.

There is no special sequence in the statements. All statements are treated equal and you can write down the commands in any order:

```
FILETYPE(FIXED,32)
INPUTFILE(c:\data\tosort.dat)
```

and

```
INPUTFILE(c:\data\tosort.dat)
FILETYPE(FIXED,32)
```

are both legal and interpreted the same by the Sort Solution parser.

Example

```
; This profile sorts the file "input.dat" into "output.dat" using
; a fixed record length of 4 byte. The file is sorted using a
; key of type LongInt and in ascending order

FILETYPE(FIXED,4)
INPUTFILE(input.dat)
OUTPUTFILE(sorted.dat)

KEY(LongInt,ASC,0,0)

; End of profile
```

Command Overview

The Sort Solution Script Language

CACHES	Defines the number of Pre-Merge caches
CHECKDRIVESPACE	Enable drive space check for the output file
COMPRESS	Allows you to compress the output file and the temporary merge files
DRIVES	Defines the drives that are used for temporary files
DRIVESPARE	Defines the minimum disk space that should remain free during the sort
FILETYPE	Defines the format of the input file
FILTER	Creates filters
HEADER	Sets options for the file header
TRAILER	Defines a trailer for the input file
INPUTFILE	Defines the name of the input file
KEY	Creates keys
LOGFILE	Defines the name of the logfile
MERGEMEM	Specifies the amount of memory available for the merge phase
RANGE	Limits the number of output records
OUTPUTFILE	Defines the name of the output file
PRIORITY	Defines the priority at which Sort Solution is executed
SORTMEM	Specifies the amount of memory available for the sort
THREADS	Defines the number of parallel threads that are used during the sort

This is an example of a full-fledged profile with all available commands. You can click on any command to go to the corresponding topic.

```
; This profile sorts the file c:\data\personal.dat to c:\data\personal.srt
; The file is sorted using four keys
; Duplicate records are removed and written to a logfile
; The input file is a DELIMITED file with an Carriage Return / Linefeed as the record delimiter,
; fields within the records are separated with semicolons

THREADS(4)
CACHES(4)
SORTMEM(512)
MERGEMEM(4096)
COMPRESS(FALSE,FALSE)
PRIORITY(NORMAL)

FILETYPE(DELIMITED,";","0x0D,0x0A")
INPUTFILE(c:\data\personal.dat)
OUTPUTFILE(c:\data\personal.srt)
LOGFILE(c:\data\doubles.dat)

KEY(StringNLS,ASC,1,0,0,LANG_GERMAN, SUBLANG_GERMAN)
KEY(StringNLS,ASC,2,0,0, LANG_GERMAN, SUBLANG_GERMAN)
KEY(UIntS,ASC,3,0,0)
KEY(Date,DESC,5,0,0,"DD.MM.YYYY")

FILTER(DUPLICATES,KEYS)
```

CHECKDRIVESPACE

Command Overview

Syntax:	CHECKDRIVESPACE(<Enable:[TRUE|FALSE]>)
Arguments:	*<Enable>* Use TRUE to enable the drive space check for the output file or FALSE to disable it
Optional:	Yes
Default:	Per default, Sort Solution performs no drive space check
Examples:	CHECKDRIVESPACE(TRUE) Enables the drive space check for the output file

Sort Solution performs by default no drive space check for the output file. If the file is to large to fit on the output drive, a »disk-full« error will occur during the sort.

If you want to check in advance if there is enough space on the output drive to hold the output file, enable the drive space check with CHECKDRIVESPACE(TRUE).

HEADER

Command Overview See also: FILETYPE , TRAILER , Examples

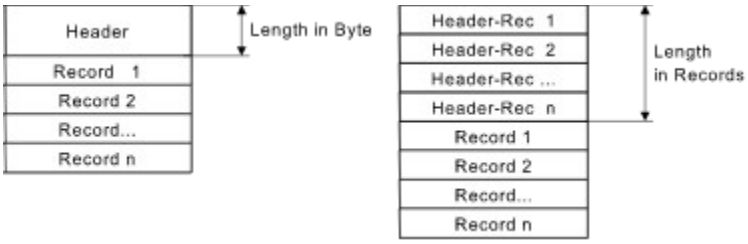
Syntax:	HEADER(<Number of Elements>,<Type:[BYTE|RECORD]>,<Keep:[TRUE|FALSE]>)
Arguments:	*<Number of Elements>* Defines the number of *Elements* in the header *<Type>* Possible values are: BYTE: The header is in bytes RECORD: The header is in records, based on the definition in the FILETYPE statement for the current profile *<Keep>* Use the value TRUE to copy the header into the output file, FALSE otherwise. If you use FALSE, the header is only skipped in the input file but *not* copied to the output file
Optional:	Yes
Default:	If no HEADER is given, the complete input file is sorted. The whole content of the input file must consist of valid records, based on the definition in the FILETYPE statement for the current profile
Examples:	HEADER(120,BYTE,TRUE) The input file contains a 120 byte long header. The header is copied to the output file HEADER(4,RECORD,FALSE) The input file contains a header consisting of 4 *records*. The header is skipped in the input file but *not* copied to the output file

Many files do contain a *header* - a sequence of bytes or records at the beginning of the file with a special meaning. This header that is not part of the »normal« records in the file.

In most cases, it makes no sense to sort the header together with the »normal« records in the file. The header should be copied into the output file but remain unchanged (and *unsorted*).

The Sort Solution HEADER statement allows you to define the size and format of the header and whether you want to copy the header into the output file.

Sort Solution supports headers that have the same record format as the input file and headers that have a completely different format. Depending on the format of the header, Sort Solution uses the unit *RECORD* or the unit *BYTE* to define the length of the header.



Header type BYTE and RECORD

If your header has the same format as the other records in the file, use the unit RECORD to describe the header. If the format of your header differs from the record format, use the unit BYTE. In all cases, you need to know the length of the header in advance.

See also: The TRAILER statement

TRAILER

Command Overview See also: FILETYPE , TRAILER , Examples

Syntax: TRAILER(<Type:[PARSE | EXPLICIT]>,<Arguments: [Expression> | Number of Bytes]>,<Keep:[TRUE|FALSE]>)

Arguments:

<Type>
Possible values are:

PARSE:	The trailer is parsed, it consists of a character sequence (*tag*) with one or more characters
EXPLICIT:	The length of the trailer is specified in bytes, from the beginning or end of the input file

<Arguments>
Depending on <Type>, this argument specifies the character sequence used to recognize the trailer or the number of bytes to skip at the end of the file.
See the comment section for more information.

<Keep>
Use the value TRUE to copy the trailer into the output file, FALSE otherwise. If you use FALSE, the trailer is only skipped in the input file but *not* copied to the output file

Optional: Yes

Default: If no TRAILER is given, the complete input file is sorted (except there is a Header defined). The whole content of the input file must consist of valid records, based on the definition in the FILETYPE statement for the current profile

Examples:

TRAILER(PARSE,"@",TRUE)
The input file is read until the '@' character is found. Only the part of the file *before* the '@' is sorted. After sorting, the trailer starting at '@' is appended to the output file

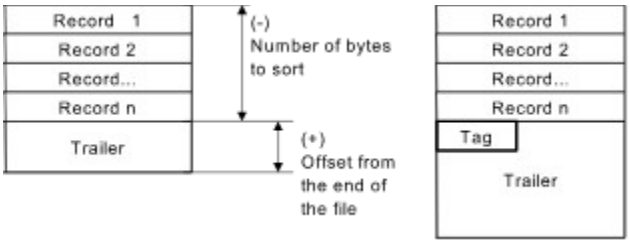
TRAILER(PARSE,".END",FALSE)
The input file contains somewhere the tag '.END'. Only the contents of the input file before this tag are sorted. The trailer starting with .END is **not** copied to the output file.

TRAILER(EXPLICIT,100,TRUE)
The trailer consists of the last 100 bytes in the file. If the file is 100.000 bytes long, only the first 99.900 bytes are sorted. The last 100 bytes in the file are treated as a trailer and are not sorted. After the sort has finished, the trailer is copied into the resulting output file.

TRAILER(EXPLICIT,**-**100000,TRUE)
A **negative** number for the trailer length specifies not the length of the trailer, but the length of the file that should be sorted. In this example, only the first 100.000 bytes of the file are sorted, independent from the actual file size. The unsorted trailer (the rest of the file) is copied to the output file after the sort.

Some files do contain a *trailer* - a sequence of bytes beginning with a special *tag* or character sequence that should not be sorted. In some cases, only a specific part of the input file should be sorted, e.g. the first 100.000 bytes and the rest of the file should remain as is. Sort Solution provides the TRAILER statement to handle such cases.

Sort Solution supports two kinds of trailers: *Explicit Trailers* and *Parsed Trailers*.



TRAILER types EXPLICIT and PARSE

Explicit Trailers

An explicit trailer is specified by a fixed offset from the beginning or end of the file. A positive size specifies the length of the trailer in bytes, counted from the end of the file. A negative number specifies the number of bytes that should be sorted, the rest of the file is treated as the trailer and ignored during the sort.

To specify a trailer that is 100 bytes long, use a statement like

TRAILER(EXPLICIT,100,TRUE)

in your profile. When you want to sort only the first *n* bytes of the input file, use a statement like this:

TRAILER(EXPLICIT, **-n**, TRUE)

where n is the number of bytes to sort. Notify the minus sign in front of the argument.

Sort Solution will issue an error message if the length you specify is bigger than the actual file size.

Parsed Trailers

A special case are files with a tag (or character sequence) that indicates the beginning of the file trailer. dBase files, for example, may contain a number of deleted records at the end of the file. dBase uses a EOF character (0x1A) to specify the end of the valid data in the file. Everything that follows the EOF should be treated as invalid.

If you want to sort a dBase file which contains deleted records at the end, you should limit Sort Solution to sort only the valid records in the file and leave the deleted records at the end of the file alone.

You can do so using a trailer statement like this:

TRAILER(PARSED, "0x1A",FALSE)

This statement will also instruct Sort Solution to skip the deleted records while producing the output file (Parameter <Keep> = FALSE).

If the tag identifying your trailer contains more than one character, you need to specify at least enough characters in the trailer statement to uniquely identify the trailer tag. For example, for a file format like this:

```
Albertt
Lucyt
Jackt
Nadt
Chuckt
Lolat
Christinet
<end>t
The trailer of the file
contains an arbitrary number of bytes...
```

the tag for the trailer is "<end>". All contents of the file following this take should be ignored during the sort.

Use a TRAILER statement like to following:

TRAILER(PARSED,"<end>",TRUE)

to skip everything that follows <end> (including the <end> itself).

Please note that parsed takes require a considerable amount of processing, since the whole input file must be scanned to identify the usable part of the file. For very large files, this can take between seconds and

minutes, depending on the speed of your machine and the throughput of your I/O subsystem.

If a file does not contain the tag specified in the profile, it is processed as if there is no TRAILER statement in the profile. This behavior is especially useful when you don't know if the file contains a trailer or not.

KEY

Command Overview See also: Keys , Key Types , Examples

Syntax:	KEY(<Typename>,[ASC|DESC], <Position>, <Offset>, <Length>, ...)
Arguments:	Variable, see Key Types
Optional:	No, each profile must contain at least one KEY statement
Default:	-
Examples:	KEY(Generic, ASC, 0,0,10) This key statement creates a key that sorts the input file after the first 10 bytes of each record in ASCending order KEY(Date,ASC,1,0,"MM/DD/YYYY") The records in the input file contain a date field at position 1. The format of the date field is specified with the mask "MM/DD/YYYY".

The KEY statement defines the sort criteria for your files. Each profile must contain at least one KEY statement, the maximum number of keys to be used in one profile is limited to 64.

Sequence of KEY statements

The sequence of the key statements in the profile defines the exact sort order. Each key has an order of precedence from top to bottom. The *n+1*th key serves as a tie-breaker for the *n*th key.

```
109889;Mainboard;124.46;
108700;Adapter;45.23
109023;Mouse;23.56;
107890;SCSI-Controller;389.00;
109023;Serial;23.56;
```

Sorting this file with the key statements

```
; Sort order defined by Name over Price
KEY(String,ASC,1,0,0)
KEY(IntS,ASC,3,0)
```

results in

```
108700;Adapter;45.23
109889;Mainboard;124.46;
109023;Mouse;23.56;
107890;SCSI-Controller;389.00;
109023;Serial;23.56;
```

But when you invert the sequence of the keys in the profile, you will get this result

```
; Sort order defined by Price over Name
KEY(IntS,ASC,3,0)
KEY(String,ASC,1,0,0)

109023;Mouse;23.56;
109023;Serial;23.56;
108700;Adapter;45.23
109889;Mainboard;124.46;
107890;SCSI-Controller;389.00;
```

For more information on how to create and use keys, please read the section Keys.

FILETYPE

Command Overview See also: Supported File Types HEADER , TRAILER , Examples

Syntax:	FILETYPE(<Type> {,<Options>})
Arguments:	*<Type>* The type (file format) of the input file. This can be one of the predefined types DELIMITED, FIXED, COUNTED, or EXPLICIT. See the table below for available options
Optional:	No, each profile must contain a FILETYPE statement
Default:	-
Examples:	FILETYPE(FIXED,32) The input file has a fixed record length of 32 byte FILETYPE(DELIMITED,";","0x0D,0x0A") The input file is in DELIMITED format. The records are of variable length and delimited with a Carriage Return / Linefeed (0x0D, 0x0A, see ASCII Table). The fields within the records are separated with a semicolon

With the FILETYPE statement you define the format of the input file. Sort Solution currently supports four different file types, each can be customized to fit your needs.

The number of arguments for FILETYPE differs, depending on the <Type> you use:

FIXED	*(FIXED,<Record length>)*	
	<Record length>	Length of the records in bytes
DELIMITED	*(DELIMITED,"<Separator>","<Delimiter>")*	
	Separator	A single, quoted character. This character is used by Sort Solution as the field separator which divides a record into fields. You can use any single character here, or an escape sequence.
	Delimiter	A *maximum of two* characters which serve as the record delimiter. The characters must be in quotes and if you use more than one character, separate them with a comma. You can use any character here or an escape sequence
COUNTED	*(COUNTED,<Number of fields>,"<Separator>")*	
	Number of fields	Number of fields that create a record.
	Separator	A single, quoted character. This character is used by Sort Solution as the field separator to divide the input file into fields. You can use any single character here, or an escape sequence.
EXPLICIT	*(EXPLICIT,<Offset>,<Length>,<Binary: [TRUE|FALSE]>)*	
	Offset	Distance, in bytes, from the beginning of the record
	Length	The length, in bytes, for the explicit record length specifier. This argument specifies how many bytes from each record are treated as the *record length specifier*.
	Binary	If you use the value TRUE here, Sort Solution uses the first two bytes of each record as the length specifier. The two bytes are treated as a 2-Byte unsigned Integer (a *Word*). The argument *Length* is ignored in this case. If you use FALSE, Sort Solutions uses *Length* bytes, starting at *Offset* from each record as the record length specifier. The length specifier must be a valid number, containing only the digits 0..9.

Escape Sequences

The ASCII table contains »normal« (printable) characters and so-called *control characters*, which have a special meaning when they are sent to a device like a printer or a console window. Control codes (or *escape codes*) are also used in files to define line breaks, page breaks and the like. DOS and Windows use for example the control characters Carriage Return and Linefeed as the line delimiter in text files.

In most of the simply text editors, it is not allowed or possible to enter control characters directly. For this reason, the Sort Solution profiles are able to handle so-called *Escape Sequences*.

Say, for example, your input file uses the control codes Carriage Return (ASCII 13decimal) and Linefeed (ASCII 10decimal) as the record delimiter. This is very common for files that are created under Windows 95 and Windows NT:

To enter these two escape codes as the delimiter in a FILETYPE statement in your profile, you must use an *escape sequence*.

First transform the decimal code of each character into it's hexadecimal equivalent, using the ASCII table in this help file:

Carriage Return (ASCII 13decimal) = 0Dhexadecimal

Linefeed (ASCII 10decimal) = 0Ahexadecimal

In a escape sequence, only hexadecimal values are allowed. Each hexadecimal value code must have a leading *0x,* so the sequence Carriage Return / Linefeed transforms to:

0x0D**,** 0x0A

When you use more than one character in an escape sequence, you must separate them with a comma.

The resulting FILETYPE statement then looks like this:

```
FILETYPE(Delimited, ";", "0x0D,0x0A")
```

If your input files comes from the UNIX environment or the Mac, it possibly uses only a Linefeed character as the record delimiter. Simply change the escape sequence for the record delimiter in the FILETYPE statement:

```
FILETYPE(Delimited, ";", "0x0A")
```

If your records a delimited with a Null (Zero) character, this escape sequence will do the job:

```
FILETYPE(Delimited, ",", "0x00")
```

When your record delimiter or field delimiter is a »normal« character which can be entered in a text editor, you don't need to use escape codes:

```
; # serves as the record delimiter
FILETYPE(Delimited, ",", "#")
```

Note: The number of characters for the field separator is limited to one character. Record delimiters can use up to two characters. This rule also applies to escape codes.

INPUTFILE

Command Overview See also: OUTPUTFILE , Examples

Syntax: INPUTFILE(<Filename>,{<Delete:[TRUE|FALSE]>})

Arguments: *<Filename>*
The name and path of the input file. If you use only a filename without a path, the file must resist in the current directory

<Delete> (Optional)
If you use the value TRUE, the input file is deleted after Sort Solution has read all records. Use FALSE or omit the flag when you don't want to delete your input file

Please note that you can use the same file as the input and output file

Optional: Yes, but then the SORTSOL.EXE command line must supply an input filename

Default: -

Examples: INPUTFILE(c:\data\input.dat)
This statement specifies the file »c:\data\input.dat« as the input file.

INPUTFILE(d:\db\log.txt,TRUE)
The file »d:\db\log.txt« is sorted and deleted after Sort Solution has read all records.

OUTPUTFILE

Command Overview See also: INPUTFILE , Examples

Syntax: OUTPUTFILE(<Filename>,{<Append:[TRUE|FALSE]>})

Arguments: *<Filename>*
The name and path of the output file. If you use only a filename without a path, the file must resist in the current directory

<Append> (Optional)
Use TRUE if you want to append the contents of the input file to an existing output file. If you use FALSE or omit the flag, an existing output file is overwritten with the contents of the input file

This option is required when you want to merge (concatenate) several input files into one common output file

Optional: Yes, but then the SORTSOL.EXE command line must supply an output filename

Default: -

Examples: OUTPUTFILE(c:\data\output.dat)
The sorted content of the input file is written to »c:\data\output.dat«. If a file with this name already exists, it is overwritten.

INPUTFILE(c:\data\output.dat,TRUE)
The sorted content of the input file is written to »c:\data\output.dat«. If a file with this name already exists, the new content is appended to the existing file

LOGFILE

Command Overview See also: FILTER , Examples

Syntax:	LOGFILE(<Filename>,{<Append:[TRUE|FALSE]>})
Arguments:	*<Filename>* The name and path of the logfile. If you use only a filename without a path, the file must resist in the current directory. *<Append> (Optional)* Use TRUE if you want to append new data to the end of an existing logfile. Use FALSE or omit the flag if you want to overwrite existing logs.
Optional:	Yes
Default:	-
Examples:	LOGFILE(c:\data\log.dat) All records filtered with a Filter are written to »c:\data\log.dat« LOGFILE(c:\data\log.dat,TRUE) New records written to the logfile »c:\data\log.dat« are appended to the end of an existing logfile. If the file does not exist, it is created.

See FILTER for more details on logfiles and their use.

FILTER

Command Overview See also: LOGFILE , Examples

Syntax:	FILTER(<Type:[DUPLICATES]>,<Options:[RECORDS|KEYS]>)
Arguments:	*<Type>* Type of the filter: Use one of the following values: DUPLICATES: Remove duplicates *<Options>)* RECORDS Remove duplicate records, binary equivalence KEYS Remove duplicate records, based on keys
Optional:	Yes
Default:	-
Examples:	FILTER(DUPLICATES,RECORDS) Removes all duplicate records from the output file. The records are compared byte by byte and only records that are binary equal are removed FILTER(DUPLICATES,KEYS) This filter removes duplicate records based on keys. Records are treated as equal if the are equal with respect to the current key definition

With the FILTER statement you have the opportunity to remove duplicate records (*Duplicates*) from your output file.

Normally, when you have a file which possible contains duplicate records, the only way to find these records is to sort the file and then browse through it and remove duplicate records manually. This is a time-consuming, error-prone and boring work to do.

With Sort Solution you can assign this task to your computer. Simply add a FILTER statement to your profile, set the appropriate options and run the sort. Only those records that are binary or key-based unique are written to the output file.

Sort Solution supports two different options for the DUPLICATES filter:

RECORDS Performs a binary comparison of all the records in your input file. Only those records that are binary equal are treated as duplicates.

KEYS Compares all records in your input file with respect to the key definition in your profile. All records with identical keys are treated as equal.

Look at the following file. The first two records are binary identical. The third record has the same name ("Max Miller"), but a different street and city name.

```
Max Miller;Sun Ave. 1313;00989;Sortal;2345,50t
Max Miller;Sun Ave. 1313;00989;Sortal;2345,50t
Max Miller;Sun Avenue 1215;00539;Sartol;6545,50t
```

If you run a filter with the option RECORDS, the first two records will be treated as duplicates because they are binary equal.

If you use the KEYS option, the result of the filter depends on your KEY definition. For example, if you use a key for the *name* field, all three records are equal, because they all have the same content: "Max Miller". If you use a second key for the *street* or the *city* field, only the first two records are treated as equal, because the third record differs in both fields.

Generally, if you want to remove duplicate records that have the same content in *n* fields, use *n* key definitions and the FILTER statement with the option KEYS. If you want to remove only records that are complete equal, use a FILTER with the option RECORDS.

RANGE

Command Overview See also: FILTER , Examples

Syntax:	RANGE(<Number of records>)
Arguments:	*<Number of records>* The number of records that should remain in the output file
Optional:	Yes
Default:	-
Example:	RANGE(10) This statement instructs Sort Solution to write only 10 records into the output file.

With the RANGE statement you can restrict the number of records that are written to the output file.

Say, for example, you want to create a *Top-10* list from the records in your input file. The only thing you have to do is to add a RANGE statement in your profile:

```
RANGE(10)
```

This statement limits the number of records written to the output file to 10. Together with the initial sort, this will create an output file containing the 10 biggest records (or smallest records, depending on your key definition).

PRIORITY

Command Overview See also: Examples

Syntax:	PRIORITY(<Priority>: [LOW|NORMAL|HIGH])
Arguments:	*<Priority>* The priority level for SORTSOL.EXE
Optional:	Yes
Default:	NORMAL
Example:	PRIORITY(LOW) Sets the priority for SORTSOL.EXE to the smallest possible level. SORTSOL.EXE works completely in the background and has a lower priority than any other process (program, application) currently running

With the PRIORITY statement you can influence how much processing power SORTSOL.EXE uses.

When you run Sort Solution on a PC that also runs other programs, e.g. a File Server or Database Server, it is sometimes desirable to limit the amount of processing power Sort Solution uses. Since sorting is a very resource-intense task, it can have a negative influence on other applications or services that are running at the same time. This is especially true for computers that must guarantee a minimum response time, like Database Servers or Internet Servers.

On the other hand, sometimes is it required that the sort of a file completes in the shortest amount of time possible.

Use the PRIORITY statement in your profile to minimize or maximize the processing power Sort Solution uses. This allows you to control the utilization of the sort engine and also the utilization of the computer that executes Sort Solution.

With the statement

```
PRIORITY(LOW)
```

you can minimize the amount of processing power for Sort Solution. The Operating System only assigns processing power to SORTSOL.EXE when no other application or service utilizes the processor. This results in a somewhat slower sort, but also leaves as much processing power as possible to the rest of the system.

The setting NORMAL is the default. When you don't include a PRIORTIY statement in your profile, Sort Solution runs on the same level as other applications.

The setting HIGH should be used only for very small files or jobs that must be finished in an extremely short amount of time.

Note: The setting HIGH has an enormous impact on the performance of your system. Sort Solution uses the whole amount of processing power that is available. Although this setting guarantees the fastest sorts possible, use this setting with care and only if it is absolutely necessary.

THREADS

Command Overview See also: CACHES , Examples

Syntax:	THREADS(<Number of threads>)
Arguments:	*<Number of threads>* Number of threads used for the sort phase
Optional:	Yes
Default:	If you don't include a THREADS statement in your profile, Sort Solution uses two threads per processor in your computer system
Example:	THREADS(4) Sort Solution uses four threads in Phase I of the sort

Sort Solution was designed and developed to make maximum use of *SMP machines* with more than one processor. Sort Solution automatically determines the number of processors in your system and uses all processors in parallel to speed up the sort phase.

If you sort very large files (50 MB and more) you can probably improve the performance of Sort Solution when you try different settings for THREADS. More on this subject can found under the topic Performance Tuning

CACHES

Command Overview See also: THREADS Examples

Syntax:	CACHES(<Number of caches>)
Arguments:	*<Number of caches>* Number of caches used for the Pre-Merge
Optional:	Yes
Default:	If you don't include a CACHES statement in your profile, Sort Solution uses 2 * <Number of Threads > caches for the Pre-Merge.
Example:	CACHES(8) Sort Solution uses 8 Pre-Merge caches during the sort phase. This will reduce the time needed for the Merge-Phase but will also increase the time needed for the initial sort.

Sort Solution uses a predefined number of Pre-Merge caches, depending on the number of processors in your system and the number of threads used.

If you sort very large files (50 MB and more) you can possibly improve the performance of Sort Solution when you try different settings for CACHES. More on this subject can found under the topic Performance Tuning

SORTMEM

Command Overview See also: MERGEMEM , THREADS Examples

Syntax:	SORTMEM(<KB>)
Arguments:	*<KB>* Amount to memory to be used by each thread during the sort phase
Optional:	Yes
Default:	If you don't use a SORTMEM statement in your profiles, Sort Solution acts as follows: First it checks how much physical memory is available in your system. If there is enough memory to sort the input file without a merge step, Sort Solution allocates a big block of memory, reads in the input file, sorts it and writes it directly to the output file. If there is not enough memory to sort the input file in one pass, Sort Solution uses 256 KB per thread and uses a final merge to create the output file. Sort Solution never uses more than 50% of the available physical memory to leave enough space for the file system and the file system cache. You can change this behavior with the SORTMEM statement
Example:	SORTMEM(2000) Sort Solution uses 2000 KB (approx. 2 MB) of memory per thead during the sort phase (Phase I)

The SORTMEM statement allows you to define how much main memory (RAM) Sort Solution uses per thread during Phase I. Normally, you should let Sort Solution decide how much memory to use. In some cases, especially when you sort files with several 100 Megabytes, you can maximize the performance of Sort Solution, when manipulate the memory requirements with the SORTMEM statement.

Please read the section Performance Tuning for more information on this subject.

MERGEMEM

Command Overview See also: SORTMEM , THREADS Examples

Syntax:	MERGEMEM(<KB>)
Arguments:	*<KB>* Amount to memory to be used during the merge phase (Phase II)
Optional:	Yes
Default:	If you don't include a MERGEMEM statement in your profiles, Sort Solution uses 2 MB RAM during the merge phase
Example:	MERGEMEM(8192) Sets the cache for Phase II to 8 MB. This can help to improve the performance when your hard disks are slow

The MERGEMEM statement defines how much memory Sort Solution uses for it's internal *file caching* mechanisms during the merge phase. Again, you should only change the default values when you need to maximize the performance for very large files. The standard setting of 2 MB file cache is absolutely sufficient for most cases.

When you hard disks are very slow, compared to the speed of your processor, you possibly can increase the throughput during the merge when you increase the amount of memory used by Sort Solution with a MERGEMEM statement in your profile.

Please read the section Performance Tuning for more information on this subject.

COMPRESS

Command Overview See also: Examples

Syntax:	COMPRESS(<Merge>: [TRUE|FALSE],<Output>: [TRUE|FALSE])
Arguments:	*<Merge>* Use the value TRUE to enable compression for the temporary files created during Phase I. Use FALSE to disable compression for temporary files *<Output>* Use the value TRUE to enable compression for the output file, or FALSE to disable the compression
Optional:	Yes
Default:	If you don't include a COMPRESSION statement in your profiles, Sort Solution uses no compression
Examples:	COMPRESS(TRUE,FALSE) Sort Solution uses the built-in compression under Windows NT to compress the temporary files created during the sort (Phase I) COMPRESS(FALSE,TRUE) Sort Solution only compresses the resulting output file, but not the temporary merge files.

Windows NT has a built-in compression feature which works for files, folders and complete disks. Sort Solution is able to use this feature to minimize the disk space needed for temporary files created during the sort and the resulting output file.

Use the COMPRESS statement in your profile to enable compression for temporary merge files, the output file, or both.

Note: The compression works only under Windows NT and on *NTFS formatted* drives.
When you use drives or folders with activated compression, the COMPRESS statement has not impact on these settings.
If you sort an already compresses file, and the output file is the same as the input file, the output file will automatically be compressed.
COMPRESS is ignored on machines that run under Windows 95.

DRIVES

Command Overview See also: DRIVESPARE, COMPRESS , Examples

Syntax:	DRIVES(<List of drive specifiers>)
Arguments:	*<List of drive specifiers>* A list of drive specifiers, separated by comma
Optional:	Yes
Default:	If you don't use a DRIVES statement in your profiles, Sort Solution will automatically use all local drives that are classified as »writeable« (No CD-ROM's, no floppies, no network drives) On all drives used for sorting, Sort Solution searches for a directory named **SORTSOL.TMP**. If it finds a directory with this name, temporary files are created in this directory. If no such directory exits, Sort Solution uses the root directory.
Examples:	DRIVES(c:, d:) Sort Solution uses the local drives C: and D: for the temporary files created during the sort DRIVES(C:) Sort Solution will only use drive C: for temporary files

When you sort large files, Sort Solution uses temporary files to store pre-merged blocks of data for the merge phase. With the DRIVES statement you can influence how Sort Solution distributes these temporary files over the drives available in your computer.

The disk space needed for all temporary files is exactly the size of the input file. If you have an input file with 100 MB, and you use two drives to store temporary files, each of the two files created will be 50 MB in size. If there is not enough disk space on one of the drives, Sort Solution will automatically distribute all remaining records into the file on the drive that has sufficient space.

For performance reasons, it is desirable to use as many drives as possible for the temporary files. Since Sort Solution is able to access all files in parallel and writes/reads records to/from all files asynchronously, the more drives are available the faster Sort Solution will run.

However, sometimes you may want to restrict the drive usage of Sort Solution to one or more specific drives. Probably, you don't have *write access* to one of your local drives or you have to make sure that a specific drive remains free for other purposes.

In these cases, you must instruct Sort Solution to use only specific drives for temporary files. Use the DRIVES statement in your profiles and specify only the drives that can be used by Sort Solution.

File Distribution

From the performance point of view, you should distribute all files that are used during the sort on as many drives as possible. This includes the input file, the output file and especially the temporary files created during Phase I. The following table gives some hints on how to establish the optimal distribution depending on how many drives you have installed.

Number Of Disks	How To Distribute
1	Input file, output file and temporary files are all on the same disk. This is the slowest possible solution

2	Use the same drive for the input file and the output file. Use the second drive to store temporary files. For example:
	
	`INPUTFILE(c:\data\input.dat)`
	`OUTPUTFILE(c:\data\sorted.dat)`
	`DRIVES(d:)`

	This results in an optimal distribution and a maximal throughput for all files
3-...	If you have more than 2 drives in your system, always try to store temporary files on the drives that are not used for the input and output file

Administration Issues

Sort Solution uses the root directory to store temporary files. If you want to avoid this for security reasons, create a folder names SORTSOL.TMP on the drives. Sort Solution will automatically use this folder to store the temporary files.

This gives you a lot more control on the disk usage, compression and quotas. You can also control access rights for your users to this directory. The user that executes Sort Solution needs the rights READ, WRITE, and DELETE for files in this directory. This allows you to limit the rights for Sort Solution users to a specific directory on the drives on your server.

If the drives are formatted with NTFS you also can enable compression for the directory SORTSOL.TMP and hence minimize the disk usage for temporary files created by Sort Solution.

Further information on how the DRIVES statement affects the overall performance of Sort Solution can be found in the section Performance Tuning.

DRIVESPARE

Command Overview See also: DRIVES , COMPRESS , Examples

Syntax:	DRIVESPARE(<MB>)
Arguments:	*<MB>* Amount of disk space that must remain free on the disks that are used for temporary files
Optional:	Yes
Default:	10 MB
Example:	DRIVESPARE(50) Sort Solution keeps a minimum of 50 MB free disk space on all drives used for temporary files

The cumulated size of all temporary files is exactly the size of the input file (except you use compression).

Sort Solution distributes the temporary files equally between all available drives. If one or more of the drives have not enough space to hold more records, Sort Solution automatically distributes the remaining records to other drives that have sufficient space.

If you need to make sure that on each drives a minimum of disk space remains free, use the DRIVESPARE statement to specify the amount that should be saved on any drive.

Performance Tuning

The overall performance that can be achieved with Sort Solution depends on a variety of factors:

- **Format of the input file**
  It is much faster to read a file with a FIXED record length than to parse a DELIMITED or COUNTED format
- **Number of keys used in parallel**
  The more keys you need to sort your file, the slower the sort will be
- **The key complexity**
  It is much faster to use a key type like Generic or LongInt than it is to use a key type like Date or KEY: Numbers as Strings , because the latter keys have to parse and analyze the field content

Besides these factors the resulting performance depends heavily on the available hardware. Sort Solution is designed to sort multi-million record files on the fastest possible speed. It uses all available resources like hard disks, memory and processor(s) to improve the overall performance of the sort.

When you sort small files up to 50 MB and about a million records, all possible improvements you can make by modifying the performance-related settings in the profile will sum up to only a few seconds compared to the default settings already built into Sort Solution. For example, it takes only about 45 seconds to sort a 1-Million 32 MB file on a Pentium-166 with rather slow disks!

But when you need to sort files which have 100 Millions or records or files that are several Gigabytes in size, you can improve the performance of Sort Solution remarkably by doing some performance testing in advance and use the results from these tests to optimize the settings in all your profiles.

The following table gives you some information on how different hardware components affect the overall performance of Sort Solution.

Component	Influence
Memory	The amount of memory available in your computer has an enormous influence on the overall system performance. The more memory you have the more memory is available for system-internal purposes like file caching. Sort Solution's memory requirements are low, compared to other sort utilities available on the market. The speed of Sort Solution depends more on the overall performance of the file system which improves if more memory is available
Hard disks and controllers	The faster your hard disks and controllers, the better is the overall performance of Sort Solution. Due to it's unique internal architecture, Sort Solution usually sorts a block of records faster than it can read or write it to or from the disks in your system. Hence the performance of Sort Solution is said to be **I/O bound**. The disk subsystem in your computer usually is the **bottleneck** when you sort large files. You can improve the overall sort performance by adding a second disk controller when you use SCSI devices and by adding one or more additional hard disks to your system
Processor(s)	The speed of your processor influences the raw sort speed of Sort Solution. This means the speed at which Sort Solution can sort records that are in memory. The faster the processor the faster Sort Solution can parse, scan and sort the records. Under normal circumstances, Sort Solution can sort records in memory faster than the can be read or written. If your processor is very fast compared to the speed of your hard disks, a faster processor will *not* improve the performance of the sort. When you have input files that need a large amount of preprocessing and parsing (e.g. DELIMITED) and you have keys that also require parsing (like Date or IntS), the performance of your processor has a higher impact the overall sort performance compared to simple file formats like FIXED with generic keys

THREADS, CACHES and SORTMEM

A thread is a unit of processing, operating independently from other threads in the system. Every thread reads, sorts, and writes a block of data in parallel with other threads. The key to an optimal sort performance is to keep the processor at a maximum performance and to utilize the disks in your system to the maximum extent possible.

Sort Solution uses per default

THREADS = *<Number or processors> * 2*

threads in the sort phase. You can experiment with this setting in your profile to fine-tune the overall performance of the sort phase.

Start with two threads per processor. If this results in a processor utilization under 75%-89%, double the number of threads until a peek performance is reached. You can view the processor utilization with the Performance Monitor under Windows NT or the System Monitor under Windows 95.

During the sort in Phase I (see Sort Solution Technical Backgrounder) Sort Solution creates blocks of presorted records, called *Runs*. To minimize the number of runs that must be merged in Phase II of the sort, a number of these presorted blocks are merged into one longer run before they are written to a temporary file on disk (Pre-Merge). If you increase the number of blocks in the Pre-Merge the sort phase gets slower but the final merge (Phase II) will run faster. You have to do some measurements to find the optimal value for CACHES.

The default used by Sort Solution is

CACHES = *<Number of threas> * 2*

This has proven as a good default setting. If you sort very large files, you can improve the performance of the Merge phase by pre-merging more blocks in the sort phase. Keep in mind that the more CACHES you use, the slower Phase I will get.

The SORTMEM statement defines how big the sorted blocks in Phase I are. If you use a SORTMEM setting of 256 KB, each thread will read a 256 KB block from the input file, sort it and write it to a temporary file. If your processor utilization is low (under 50%), increase SORTMEM in steps of 256 KB until the processor is at least utilized about 80%.

Principally, the smaller the blocks in Phase I, the faster the sort will run, but the slower the final merge will be. The performance of the merge depends heavily on how many blocks (Runs) of presorted data have to be merged.

You should try to minimize the number of runs created in Phase I using the CACHES and SORTMEM settings, but without slowing down Phase I two much.

MERGEMEM

The MERGEMEM statement lets you define how much memory Sort Solution uses for it's internal file cache during the merge in Phase II. The default for MERGEMEM is 2 MB which is sufficient for the most cases. Don't set this parameter too high, because the memory you assign to Sort Solution's internal caches cannot longer be used by the built-in file system cache of Windows. When you assign to much memory to Sort Solution, the overall system performance degrades because Windows has not enough memory left to do proper file caching.

If your machine has 64 - 128 KB of RAM, you can experiment with MERGEMEM settings up to 16 MB. When the merge gets slower after you increased the MERGEMEM setting, reduce it back to the last tested value.

Rules of Thumb

- Use two threads per processor. If you sort files with small records, use 256 to 512 KB per thread.
  If you sort files with a record size of about 50 bytes and more, increase the memory per thread to 1 - 4 MB
- Experiment with the number of CACHES. Start with two caches per thread and do some performance measurements. Then increase the number of CACHES by factors of 2 until the overall performance (Phase I and Phase II) drops.
  You can also try to vary the SORTMEM setting to increase the block size per thread. This usually results in a faster sort on fast processors
- When your system has at least 64 MB or RAM or more, try to increase the MERGEMEM setting to give Sort Solution more memory to work with during the merge. If the performance of the sort decreases after you have increased the MERGEMEM setting, this is a sign of a low-memory condition for the Operating System cache. In this case, reduce the MERGEMEM setting back to the original value

If the hard disks in your computer work very hard, but the processor utilization goes down to only a few

percent, this is a sign of *trashing*. Thrashing means that the Operating System has not enough physical memory available to fulfill all requests and needs to swap memory to and from the disks at a high ratio. This results in a heavy decrease in performance for Sort Solution. In this case, close some applications and reduce the amount of memory needed for sorting with the SORTMEM and MERGEMEM settings.

Automatic Optimization

Sort Solution has a built-in optimization strategy which works very well in most cases. Therefore you normally don't need to fiddle around with THREADS, CACHES, SORTMEM, and MERGEMEM. Sort Solution will use the optimal settings automatically.

In some rare cases, e.g. when you sort files with small records (16 bytes an less), this automatic optimization can fail and Sort Solution will use to much memory during Phase I which will result in a decreased sort performance.

In these cases, use a SORTMEM statement in your profile with a maximum of 256 KB to limit the amount of memory used by Sort Solution:

```
SORTMEM(256)
```

This will reduce the memory consumption to a normal level and results in an overall increased performance and processor utilization.

Analyzing the Sort Statistics

SORTSOL.EXE dumps some statistical information after each run onto the screen. You can use this information to optimize the performance of Sort Solution.

```
Sort(MS)..: 100%
Merge.....: 100%

Input file                    : C:\IN\1M_20.DAT
Output file                   : C:\OUT\SORT.OUT
Log file                      :
Input Filesize                : 20,000,000 Bytes
Records processed             : 1,000,000
Records filtered:             : 0
Time to complete              : 55 s (Sort: 27 s, Merge: 27 s)
  Avg. block time for load    : 412 ms
  Avg. block time for sort    : 963 ms
  Avg. block time for pre-merge : 1573 ms
Number of runs                : 10
Cache per run                 : 204 KB
```

Timings for Sort and Merge

If the merge takes much longer than the sort, you should try to reduce the number or runs. This can be done with the CACHES statement or the SORTMEM statement. First try to increase the size of the blocks that are processed in Phase I by increasing SORTMEM. Alternatively you can also double the number of CACHES. Both settings affect the number of runs created during the sort and therefore the number of runs that must be merged in Phase II.

»Avg. block time for load« und » Avg. block time for sort«

These two timings show the time in milliseconds that is required to read, parse and sort a block in Phase I. The optimal performance is reached when both timings are nearly identical. If the time for reading and parsing a block is smaller than the time for the sort, reduce the size of the blocks with the SORTMEM statement. If the sort is faster than the read and parse, you can increase the block size.

»Avg. block time for pre-merge«

This is the time that is needed to merge and write CACHES blocks of presorted data. The key to a maximum performance is to keep this timing as low as possible without increasing the number of runs to much. If you use a large number of CACHES, this timing gets slower, but the merge gets faster. If you use a smaller setting for

CACHES, the overall time for Phase I decreases but the merge phase gets slower. Try to find the best value for CACHES without making Phase I slower than Phase II.

A Final Word

Sort Solution was designed to work nearly optimal under all conditions. The defaults for all performance-related settings have been chosen with care to guarantee an optimal performance.

If you sort files that are only several megabytes in size, there is not much room for improvement by modifying the default settings.

On the other hand, when you have to sort files with several gigabytes in size, you can achieve a noticeable performance improvement by adjusting Sort Solution to the characteristics of your computer system and the kind of files you are sorting.

Incorporating Sort Solution into Your Applications

Using Sort Solution with C/C++ Using Sort Solution with Visual Basic
The Sort Solution ActiveX Control

Sort Solution exposes an API (*Application Programming Interface*) with only a few functions that allow you to access the whole functionality that is built into SORTSOL.DLL. You should be able to include Sort Solution into your application within only a few minutes.

Files Needed

Sort Solution comes in one single self-contained DLL that needs no additional runtime libraries. The required interface files for C/C++ and Visual Basic are included in the Sort Solution distribution package.

C/C++	
sortsol.dll	The Sort Solution DLL
sortsoli.h	Interface files for C/C++
sortsol.lib	Import library to be used with C/C++
Visual Basic	
sortsol.dll	The Sort Solution DLL
sortsoli.bas	The interface file for Visual Basic
ActiveX	
sortsolx.ocx	The Sort Solution ActiveX control
sortsol.dll	The Sort Solution DLL
sortsolx.lic	The license file (only needed during development, don't distribute this file to your customers)

Distributing SORTSOL.DLL

If you have purchased a license for Sort Solution, you are entitled to distribute SORTSOL.DLL together with your applications without any additional fee. You are not allowed to change the name of the DLL or the copyright information contained within the file.

Using Sort Solution with Multiple Clients

Sort Solution is able to handle sort requests from several clients at the same time in parallel. It makes no difference if these requests come from one application or from different applications. The Sort Solution DLL is completely thread-safe.

The API

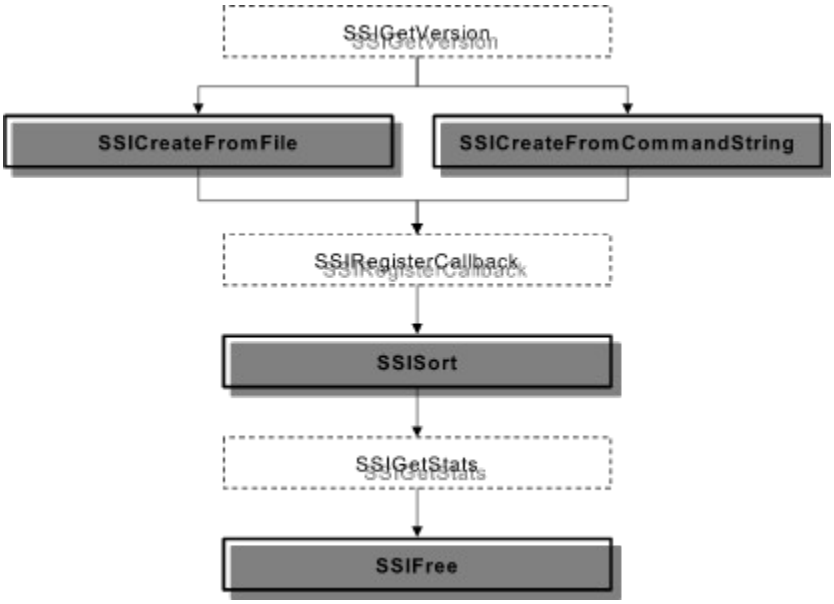
Sort Solution uses *sort instances* to handle requests from several clients. After an instance has been created, all further requests to this instance are made using a *handle* that is returned from the function that creates the instance.

The following table contains an overview of all Sort Solution API functions together with a short description of each of the functions.

SSICreateFromFile	This function takes the name of the Sort Solution profile and returns a handle to a new instance created from that profile
SSICreateFromCommandString	This function takes a string containing statements from the Sort Solution Script Language and returns a handle to a new instance
SSISort	Executes the sort for a handle created with *SSICreateFromFile* or *SSICreateFromCommandString*
SSIFree	Frees all allocated resources associated with an instance

SSIRegisterCallback	Registers a callback function that can be used to provide feedback on the sort progress in your application
SSIGetStats	Returns statistical information about a specific sort instance
SSIGetErrorMessage	This function can be used to retrieve further information for all Sort Solution error codes
SSIGetVersion	Returns the version of the SORTSOL.DLL. This function can be used to adjust to different versions of SORTSOL.DLL

The following picture shows the sequence of Sort Solution function calls within an application. The dotted execution paths are optional, only the functions shown in gray are required to sort a file.



The Sort Solution API

Further topics:

Structures

Error Handling

Analyzing Statistics

Using Sort Solution with C/C++

Using Sort Solution with Visual Basic

The Sort Solution ActiveX Control

Structures

Using Sort Solution with C/C++ Using Sort Solution with Visual Basic

The Sort Solution API declares a number of constants and structures that are used with the API functions.

Declarations for C/C++ (sortsoli.h)

```
typedef struct tagSORTSOL_CMDFILESTATUS
{
  unsigned int  uSize;
  int           LineNo;
  char          ErrLine[256];
  BOOL          Override;
  TCHAR         InputFileName[MAX_PATH];
  BOOL          DeleteInput;
  TCHAR         OutputFileName[MAX_PATH];
  BOOL          AppendOutput;
  TCHAR         LogFileName[MAX_PATH];
  BOOL          AppendLog;
  DWORD         PriorityClass;
} SORTSOL_CMDFILESTATUS;


typedef struct tagSORTSOL_STATS
{
  unsigned int   uSize;
  LARGE_INTEGER  liBytesSorted;
  DWORD          dwSortTime;
  DWORD          dwMergeTime;
  DWORD          dwAvgBlockLoadTime;
  DWORD          dwAvgBlockSortTime;
  DWORD          dwAvgBlockMergeTime;
  LARGE_INTEGER  liRecordsProcessed;
  LARGE_INTEGER  liRecordsFiltered;
  DWORD          dwNumberOfRuns;
  DWORD          dwCachePerRun;
} SORTSOL_STATS;
```

Declarations for Visual Basic (sortsoli.bas)

```
Public Type SORTSOL_CMDFILESTATUS
  Size As Long
  LineNo As Long
  ErrLine As String * 256
  Override As Long;
  InputFileName As String * 260
  DeleteInput As Long
  OutputFileName As String * 260
  AppendOutput As Long
  LogFileName As String * 260
  AppendLog As Long
  PriorityClass As Long
End Type


Public Type SORTSOL_STATS
  Size As Long
  BytesSortedLo As Long
  BytesSortedHi As Long
```

```
  SortTime As Long
  MergeTime As Long
  AvgBlockLoadTime As Long
  AvgBlockSortTime As Long
  AvgBlockMergeTime As Long
  RecordsProcessedLo As Long
  RecordsProcessedHi As Long
  RecordsFilteredLo As Long
  RecordsFilteredHi As Long
  NumberOfRuns As Long
  CachePerRun As Long
End Type
```

SORTSOL_CMDFILESTATUS

This structure is used with the *SSICreateXXX*...functions. It contains arguments for the functions on call and is filled with the results of the functions on return.

Items marked with * are used to override settings from the profile. See the remarks below.

uSize	This element contains the size of the structure in byte and must be set **before** the function call. The Sort Solution API uses this member to handle different versions of the structure (for future releases)
LineNo	Contains the line number where an error occurred on return or 0 when the operation completed successfully
ErrLine	Contains the full text of the line where the error occurred or and empty string when the operation completed successfully
Override	If this flag is set to TRUE, the items marked with * are used to override settings from the profile. See remarks below
InputFileName*	Returns the name of the input file. This name is extracted from the INPUTFILE statement in the profile. If the flag *Override* is set to TRUE, this field overrides the INPUTFILE statement in the profile if it is not NULL (empty)
DeleteInput*	This flag is set to TRUE if the input file is deleted after the sort. This flag is the equivalent to the <Delete> argument in the INPUTFILE statement. If the flag *Override* is set to TRUE in advance, this flag overrides the <Delete> argument of the INPUTFILE statement in the profile. **It is important** that you set this flag to FALSE when you use *Override* to avoid an unintended deletion of the input file.
OutputFileName*	Returns the name of the output file. This name is extracted from the OUTPUTFILE statement in the profile. If the flag *Override* is set to TRUE, this field overrides the OUTPUTFILE statement in the profile if it is not NULL (empty)
AppendOutput*	This flag is set to TRUE on return if the OUTPUTFILE statement in the profile has the <Append> argument set to TRUE. If the flag *Override* is set to TRUE in advance, this flag overrides the <Append> argument of the OUTPUTFILE statement in the profile
LogFileName*	Returns the name of the logfile, if any. This name is extracted from the LOGFILE statement in the profile. If the flag *Override* is set to TRUE, the content of this field overrides the LOGFILE statement in the profile if it is not NULL (empty)
AppendLog*	This flag is set to TRUE on return if the OUTPUTFILE statement in the profile has the <Append> argument set to TRUE. If the flag *Override* is set to TRUE in advance, this flag overrides the

	<Append> argument of the LOGFILE statement in the profile
PriorityClass	This item contains one of the following values on return: 32 : NORMAL priority 64: LOW priority 128: HIGH priority It is extracted from the PRIORITY statement in the profile. If no priority is specified in the profile, NORMAL is returned

SORTSOL_STATS

Sort Solution gathers statistical information during all sort phases. This information can be retrieved using the function SSIGetStats together with this structure.

uSize	This element contains the size of the structure in byte and must be set **before** the function call. The Sort Solution API uses this member to handle different versions of the structure (for future releases)
liBytesSorted	Number of bytes sorted (size of the input file)
VB: BytesSortedLo **VB:** BytesSortedHi	For Visual Basic, this 64-Bit value is returned in two 32-Bit values
dwSortTime	Time for the sort (Phase I) in milliseconds (ms)
dwMergeTime	Time for the merge (Phase II) in milliseconds
dwAvgBlockLoadTime	Average time in ms to load a block of data (SORTMEM bytes)
dwAvgBlockSortTime	Average time in ms to sort a block of data
dwAvgBlockMergeTime	Average time to merge CACHES blocks of data (Pre-Merge)
liRecordsProcessed	Number of records sorted
VB: RecorsdProcessedLo **VB:** RecordsProcessedHi	For Visual Basic, this 64-Bit value is returned in two 32-Bit values
liRecordsFiltered	Records skipped due to a FILTER statement
VB: RecordsFilteredLo **VB:** RecordsFilteredHi	For Visual Basic, this 64-Bit value is returned in two 32-Bit values
dwNumberOfRuns	Number of runs needed for the merge
dwCachePerRun	Size of the cache per run in byte (see MERGEMEM)

Remarks

The structure SORTSOL_CMDFILESTATUS works in both directions: It holds the input arguments for the *SSICreate...* functions and also holds the results from these functions on return.

In most cases, it is sufficient to fill the whole structure with zeros and to initialize the *uSize* (Visual Basic: *Size*) member of the structure with the size of the structure in bytes:

C/C++

```
SORTSOL_CMDFILESTATUS cmdstat;
memset(&cmdstat,0x0,sizeof(cmdstat);
cmdstat.uSize = sizeof(cmdstat);

SSICreate(...)
...
```

Visual Basic

```
Dim cmdstate As SORTSOL_CMDFILESTATUS
cmdstate.Size = Len(cmdstate)
cmdstate.Override = False
```

```
stats.Size = Len(stats)

SSICreate(...)
...
```

Overriding Settings from the Profile

If you use SSICreateFromFile to run a profile from within your application, you can use the SORTSOL_CMDFILESTATUS structure to override some of the settings in the profile. This can become very handy if you want to use a set of standard profiles but set the name of the input and output file at runtime.

If you want to override the INPUTFILE, OUTPUTFILE, OR LOGFILE statements in the profile, you have to follow these steps:

- Set the member *Override* to TRUE
- Initialize one or all of the members *InputFileName, OutputFileName, LogFileName*
- Initialize the members *DeleteInput, AppendOutput, and AppendLog* to TRUE or FALSE, depending on your requirements

The call the *SSICreateFromFile* function.

If you set the member *Override* to FALSE, Sort Solution ignores these members on input and fills them on return with the information found in the profile.

Principally, you can also use the members in SORTSOL_CMDFILESTATUS to override settings in a command string used by SSICreateFromCommandString, but since this string is built on runtime anyway, it is easier to put the required arguments directly into the command string.

Examples

These statements override the settings for INPUTFILE and OUTPUTFILE in the profile with different filenames. If the profile does not contain these statements, this kind of initialization is mandatory.
Additionally the member *AppendOutput* is set to true to append the sorted content of the input file to an existing output file.

C/C++

```
SORTSOL_CMDFILESTATUS cmdstat;
memset(&cmdstat,0x0,sizeof(cmdstat);
cmdstat.uSize = sizeof(cmdstat);

cmdstat.Override = TRUE;
strcpy(cmdstat.InputFileName,"c:\\input\\sales.txt");
strcpy(cmdstat.OutputFileName,"c:\\input\\sorted.txt");
cmdstat.AppendOutput = TRUE;

SSICreate(...)
...
```

Visual Basic

```
Dim cmdstate As SORTSOL_CMDFILESTATUS
cmdstate.Size = Len(cmdstate)
cmdstate.Override = False
stats.Size = Len(stats)

cmdstat.InputFileName = "c:\input\sales.txt"
cmdstat.OutputFileName = "c:\input\sorted.txt"
cmdstat.AppendOutput = True

SSICreate(...)
...
```

Callbacks

Using Sort Solution with C/C++Using_Sort_Solution_with_C_C Using Sort Solution with Visual BasicUsing_Sort_Solution_with_Visual_Basic

The Sort Solution API supports callbacks to allow an asynchronous communication between the sort engine and the calling application.

Callbacks are supported for C/C++, Visual Basic 5.x, and all other programming environments that support the callback paradigm (e.g. Delphi, C++ Builder).

Callbacks can be used to display progress information about the current sort within an application and to abort a running sort on a user request.

To use a callback function, your application must call SSIRegisterCallback before you call SSISort. A callback is attached to a specific sort instance and reacts only on callback messages of this specific instance. If you use more than one sort instance in your application, you can also use different callbacks for each of these instances.

If a sort instance is closed with the SSIFree function, the link to the callback function is also closed. You need to re-register the callback for each sort instance you create in your application.

The callback function is declared as:

For C/C++ (sortsoli.h):

```
BOOL SortCallback(UINT Code, DWORD StatusData, DWORD Extra);
```

For Visual Basic (sortsoli.bas):

```
Public Function SortCallback(ByVal Code As Long, ByVal StatusData As Long, ByVal
Extra As Long) As Long
```

Both functions take the same arguments:

- *Code* contains the notification code for the callback
- *StatusData* contains additional info, depending on Code
- The argument *Extra* contains the application-specific value which was defined with the SSIRegisterCallback function. This argument allows you to communicate an application-defined value to the callback function. This can be a scalar value or a pointer to any kind of data that you want to use in the callback function

The callback function returns a boolean (32-Bit Integer) value set to TRUE (<> 0) or FALSE (0).

Aborting a Sort Instance

If the callback returns FALSE, the current sort is aborted and SSISort returns immediately. This allows your application to abort a running sort at any time. Please keep in mind that it can take a few seconds to abort a sort, depending on the current state of the sort engine.

The following table lists all notification codes for the callback function. These codes are delivered in the Code argument of the callback function.

Code	Description
SORTSOL_NOTIFY_SORTPERCENTAGE	This notification is sent to indicate progress for Phase I of the sort. *StatusData* contains a value between 0 and 100 which indicates the percentage of completion for the sort phase
SORTSOL_NOTIFY_MERGEPERCENTAGE	This notification is sent to indicate progress for Phase II of the sort. *StatusData* contains a value between 0 and 100 which indicates the percentage of completion for the merge phase.

	This notification is only used if a merge phase is needed (see below)
SORTSOL_NOTIFY_BEGINSORT	This notification is sent when the sort begins. The parameter *StatusData* contains one of the following values: SORTSOL_STATUS_ONEPHASESORT (One-Phase Sort) or SORTSOL_STATUS_MERGESORT (Two-Phase Sort with Merge)
SORTSOL_NOTIFY_FINISHSORT	Sent at the end of Phase I
SORTSOL_NOTIFY_BEGINMERGE	Sent at the begin of Phase II
SORTSOL_NOTIFY_FINISHMERGE	Sent at the end of Phase II
SORTSOL_NOTIFY_FINISHED	This is the last notification. Is it send immediately before SSISort returns

Example for a Callback in C/C++

```
BOOL CALLBACK SortCallback(UINT Code, DWORD Status, DWORD Extra)
{
  switch (Code)
  {
    case SORTSOL_NOTIFY_SORTPERCENTAGE:
    cout "Sort: "<< StatusData << "%" << endl;
    break;

    case SORTSOL_NOTIFY_MERGEPERCENTAGE:
    cout "Merge: " << StatusData << "%" << endl;
    break;

    case SORTSOL_NOTIFY_BEGINSORT:
    switch (Status) {
      case SORTSOL_STATUS_ONEPHASESORT:
      cout << "One-Phase Sort" << endl;
      break;

      case SORTSOL_STATUS_MERGESORT:
      cout << "Two-Phase Sort" << endl;
      break;
    }

    break;

    case SORTSOL_NOTIFY_FINISHSORT:
    cout << "Sort finished" << endl;
    break;

    case SORTSOL_NOTIFY_BEGINMERGE:
    cout << "Merge started" << endl;
    break;

    case SORTSOL_NOTIFY_FINISHMERGE:
    cout << "Merge finished" << endl;
    break;

    case SORTSOL_NOTIFY_FINISHED:
    cout << "Sort completed" << endl;
    break;
  }

  // Continue sorting
```

```
  return TRUE;
}
```

Example for a Callback in Visual Basic (shortened, see above)

```
Public Function SortCallback(ByVal Code As Long, ByVal StatusData As Long, ByVal
Extra As Long) As Long
  Select Case Code
  Case SORTSOL_NOTIFY_SORTPERCENTAGE
  ' ...
  Case SORTSOL_NOTIFY_MERGEPERCENTAGE
  ' ...
  End Select

  ' Continue sorting
  SortCallback = 1
End Function
```

Error Handling

Error Messages Using Sort Solution with C/C++ Using Sort Solution with Visual Basic

The Sort Solution API differentiates between to kinds of errors:

- Syntax errors or logical errors in the profile or the command string used with SSICreateFromFile or SSICreateFromCommandString.
- Runtime errors that occur during the sort, e.g. »disks full« or »invalid records in input file«

All Sort Solution API functions do return a SORTSOL_ERROR (Visual Basic: *Long*) value which contains a result code. This result code indicates success or an error condition.

The declaration files for C/C++ and Visual Basic contain constants for all error conditions that are defined for the Sort Solution API.

The code SOSOERR_SUCCESS stands for operation successfully completed, all other SOSOERR_xxx codes indicate some kind of warning or error.

You can use the SSIGetErrorMessage function to retrieve a textual description (in English) from the Sort Solution API for each of the SOSOERR_xxx codes.

Don't forget to check the return code of each of the API functions on return!

Example for C/C++

```
SORTSOL_ERROR result;

result = SSICreate(...);
if (result != SOSOERR_SUCCESS) {
  // Handle the error here...
  char msg[256];
  unsigned len = sizeof(msg);
  SSIGetErrorMessage(result,msg,&len);
  printf("\n%s\n",msg);
}
```

Example for Visual Basic

```
Dim result As Long
Dim msg As String

result = SSICreate(...)
If (result <> SOSOERR_SUCCESS) Then
  ' Handle the error here
  msg = String$(255, 0)
  result = SSIGetErrorMessage(result, msg, 255)
  MsgBox msg, , "Sort Solution Error"
End If
```

Analyzing Statistics

See also: Structures

You can analyze the statistics gathered during the sort by using the SSIGetStats. This functions fills a structure of type SORTSOL_STATS with the same statistical information that is displayed by SORTSOL.EXE after he sort.

C/C++

```
SSIH  handle;

SORTSOL_STATS stats;
memset(&stats,0x0,sizeof(stats));
stats.uSize = sizeof(stats);

SSICreate(handle,...)
SSISort(handle)

SSIGetStats(handle,&stats);

SSIFree(handle)
```

Visual Basic

```
Dim stats As SORTSOL_STATS
stats.Size = Len(stats)

SSICreate(handle,...)
SSISort(handle,...)

SSIGetStats(handle,&stats)

SSIFree(handle,...)
```

Handling 64-Bit Data Types

Sort Solution uses 64-Bit Integers for several members of the structures defined in the Sort Solution API. These data types hold file sizes, the number of sorted records and all other values than can get larger than 32-Bit.

The C/C++ API of Sort Solution uses the union LARGE_INTEGER defined by the Windows API (windows.h):

```
typedef union _LARGE_INTEGER {
  struct {
    DWORD LowPart;
    LONG HighPart;
   };
   LONGLONG QuadPart;
} LARGE_INTEGER;
```

The member *QuadPart* of this structure holds the 64-Bit value, the two members *LowPart* and *HighPart* hold the lower and higher 32-Bit of *QuadPart*.

Depending on the programming language and the development environment you are using, you can use member *QuadPart* directly or you must stick to the two 32-Bit members of the structure.

The declaration file »sortsoli.bas« for Visual Basic declares for each 64-Bit type in the Sort Solution API two 32-Bit types, which can be safely used together with Visual Basic:

```
BytesSortedLo As Long
BytesSortedHi As Long
RecordsProcessedLo As Long
RecordsProcessedHi As Long
RecordsFilteredLo As Long
```

```
RecordsFilteredHi As Long
```

If you are sure that the files you are sorting never exceed the 2 Gigabyte limit of a signed 32-Bit integer and the number of records is always less then 2 Billion records, it is safe to use only the lower 32 bit of each of the 64-Bit data members.

See also: Structures

SSICreateFromFile

See also: SSICreateFromCommandString

The function *SSICreateFromFile* creates a new sort instance from an existing profile.

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSICreateFromFile(SSIH* pHandle, LPCTSTR Filename,
SORTSOL_CMDFILESTATUS* pStatus);
```

Declaration for Visual Basic

```
Public Declare Function SSICreateFromFile (ByRef handle As Long, ByVal Filename As
String, ByRef cmdstate As SORTSOL_CMDFILESTATUS) As Long
```

Arguments

C/C++	VB	Description
pHandle	handle	This variable contains the handle of the created sort instance on return You must use this handle for all subsequent operations on the new instance
Filename	Filename	Full qualified name of the profile
pStatus	cmdstate	A reference (VB) or pointer (C/C++) to a variable of type SORTSOL_CMDFILESTATUS. This structure is uses to set extended arguments and to return information from the function

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

See Using Sort Solution with C/C++ or Using Sort Solution with Visual Basic for a complete example for this function.

SSICreateFromCommandString

See also: SSICreateFromFile

The function *SSICreateFromCommandString* creates a new sort instance from a command string. The string must conform to the rules for Sort Solution profiles.

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSICreateFromCommandString(SSIH* pHandle, LPCTSTR Commands,
SORTSOL_CMDFILESTATUS* pStatus);
```

Declaration for Visual Basic

```
Public Declare Function SSICreateFromCommandString (ByRef handle As Long, ByVal
command As String, ByRef cmdstate As SORTSOL_CMDFILESTATUS) As Long
```

Arguments

C/C++	VB	Description
pHandle	handle	This variable contains the handle of the created sort instance on return You must use this handle for all subsequent operations on the new instance
Commands	command	This string contains valid Sort Solution statements to be executed by Sort Solution. The format of the string must match the format of a Sort Solution profile: one statement per line, each line delimited with a Carriage Return / Linefeed
pStatus	cmdstate	A reference (VB) or pointer (C/C++) to a variable of type SORTSOL_CMDFILESTATUS. This structure is uses to set extended arguments and to return information from the function

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

Sample for C/C++

```
char[] commands =
"INPUTFILE(c:\\data\\tosort.dat)\r\n"\
"OUTPUTFILE(c:\\data\\sorted.dat)\r\n"\
"FILETYPE(FIXED,45)\r\n"\
"KEY(Generic,ASC,0,0,10)";

SORTSOL_ERROR result = SSICreateFromCommandString(&handle,commands,&cmdstat);
...
```

Sample for Visual Basic

```
Dim commands As String
Dim result as Long

commands = "INPUTFILE(c:\data\tosort.dat)" & Chr(13) & Chr(10) & _
           "OUTPUTFILE(c:\data\sorted.dat)" & Chr(13) & Chr(10) & _
```

```
            "FILETYPE(FIXED,45)" & Chr(13) & Chr(10) & _
            "KEY(Generic,ASC,0,0,10)"

result = SSICreateFromCommandString(handle,commands,cmdstat)
```

SSISort

See also: SSICreateFromFile SSICreateFromCommandString

The function *SSISort* executes a sort instance prepared with SSICreateFromFile or SSICreateFromCommandString.

If you want to register a callback, you must do this before you call *SSISort*.

SSISort returns to the caller after the sort has finished. You can use a callback to keep your application responsive during this time or you can put the call to *SSISort* into an special worker thread in your application if this option is supported by your development environment.

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSISort(SSIH Handle);
```

Declaration for Visual Basic

```
Public Declare Function SSISort (ByVal handle As Long) As Long
```

Arguments

C/C++	VB	Description
Handle	handle	A handle to a sort instance created with one of the SSICreatexxx functions

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

See Using Sort Solution with C/C++ or Using Sort Solution with Visual Basic for a complete example on how to use this function.

SSIFree

See also: SSICreateFromFile SSICreateFromCommandString SSIFreeSSIFree

The function *SSIFree* frees all resources and temporary files allocated by a sort instance. Every handle created with one of the *SSICreatexxx* functions must be freed with *SSIFree*.

If you call *SSIFree* on a sort instance which is currently executing, the sort is aborted and all resources and temporary files are freed. See also Aborting a Sort Instance.

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSIFree(SSIH Handle);
```

Declaration for Visual Basic

```
Public Declare Function SSIFree (ByVal handle As Long) As Long
```

Arguments

C/C++	VB	Description
Handle	handle	A valid handle to the sort instance which should be freed

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

See Using Sort Solution with C/C++ or Using Sort Solution with Visual Basic for a complete example on how to use this function.

SSIRegisterCallback

The function *SSIRegisterCallback* creates a callback for a specific sort instance. See Callbacks for more information on callbacks.

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSIRegisterCallback(SSIH Handle, SORTSOL_NOTIFYCALLBACK
lpCallback, DWORD Extra);
```

Declaration for Visual Basic

```
Public Declare Function SSIRegisterCallback (ByVal handle As Long, ByVal Callback As
Long, ByVal Extra As Long) As Long
```

Arguments

C/C++	VB	Description
Handle	handle	A valid handle to a sort instance
lpCallback	Callback	Address of the callback function
Extra	Extra	A application-defined value. Sort Solution routes this value to the callback function at every call. You can use this freely to pass some application defined value to the callback function

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

Example for C/C++

```
BOOL CALLBACK SortCallback(UINT Code, DWORD Status, DWORD Extra)
{
  switch (Code)
  {
  case SORTSOL_NOTIFY_SORTPERCENTAGE:
  break;
  // ... Other codes
  }

  return TRUE;
}

...

SSICreateXXX(...)
SSIRegisterCallback(handle,SortCallback,0);
SSISort(...)
...
```

Example for Visual Basic

```
Public Function MySortCallback(ByVal Code As Long, ByVal StatusData As Long, ByVal
```

```
Extra As Long) As Long
  Select Case Code
  Case SORTSOL_NOTIFY_SORTPERCENTAGE
  ' ... Other codes
  End Select

  SortCallback = 1
End Function

...

SSICreateXXX(...)
SSIRegisterCallback(handle, AddressOf SortCallback, 0)
SSISort(...)
...
```

See the examples in the \Samples folder of your Sort Solution installation for more details on how to use callbacks. The two examples for Visual C++ and Visual Basic use callbacks to provide user feedback during the sort.

SSIGetStats

Analyzing Statistics

The function *SSIGetStats* returns statistical information that Sort Solution gathers per sort instance. Please see Analyzing Statistics for more details.

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSIGetStats(SSIH Handle, SORTSOL_STATS* pStats);
```

Declaration for Visual Basic

```
Public Declare Function SSIGetStats (ByVal handle As Long, ByRef stats As
SORTSOL_STATS) As Long
```

Arguments

C/C++	VB	Description
Handle	handle	Handle for a valid sort instance. The *SSISort* function must have been run on this handle *before* you cal *SSIGetStats*
pStats	stats	A reference or pointer to a variable of type SORTSOL_STATS. The function fills this variable with the statistical information

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

Examples

See Analyzing Statistics for an example on how to use this function.

SSIGetErrorMessage

Error Handling Error Messages

The function *SSIGetErrorMessage* is used to get a descriptive message for any Sort Solution error code.

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSIGetErrorMessage(SORTSOL_ERROR ErrorCode, LPTSTR lpBuffer,
DWORD* nSize);
```

Declaration for Visual Basic

```
Public Declare Function SSIGetErrorMessage (ByVal ErrorCode As Long, ByVal Text As
String, ByRef Size As Long) As Long
```

Arguments

C/C++	VB	Description
ErrorCode	ErrorCode	The error code for which a text should be returned
lpBuffer	Text	Pointer to a buffer for the message
nSize	Size	Number of characters that can be written into *lpBuffer* (Text). In C/C++ this includes the terminating '\0'.

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

Example for C/C++

```
SORTSOL_ERROR result;

result = SSICreate(...)

if (result != SOSOERR_SUCCESS) {
  char msg[255];
  SSIGetErrorMessage(result,msg,sizeof(msg));
}
```

Example for Visual Basic

```
Dim result As Long
Dim msg As String

result = SSICreate(...)

If (result <> SOSOERR_SUCCESS) Then
    msg = String$(255, 0)
    result = SSIGetErrorMessage(result, msg, 255)
End If
```

SSIGetVersion

The function *SSIGetVersion* returns information about the version of the Sort Solution DLL. You can use this information to handle different versions of Sort Solution that might be installed on your customers PC's.

A version number is composed from four values of type WORD (unsigned short int). These values are packed in two variables of type DWORD (VB: Long).

Declaration for C/C++

```
SORTSOL_ERROR WINAPI SSIGetVersion(DWORD* pVNMS, DWORD* pVNLS);
```

Declaration for Visual Basic

```
Public Declare Function SSIGetVersion (ByRef MS As Long, ByRef LS As Long) As Long
```

Arguments

C/C++	VB	Description
pVNMS	MS	A DWORD containing the version number. The HIWORD specifies the most significant 32 bits of the file's binary version number. The LOWORD specifies the least significant 32 bits of the file's binary version number
pVNLS	LS	A DWORD containing the build number. The HIWORD specifies the most significant 32 bits of the file's build number. The LOWORD specifies the least significant 32 bits of the file's build number

Function Result

The function returns SOSOERR_SUCCESS if the operation completes successfully or one of the other SOSOERR_xxx codes otherwise.

See Error Handling for more details.

Example for C/C++

```
DWORD lo = 0;
DWORD hi = 0;
SSIGetVersion(&hi,&lo);

printf("Version: %u.%u, Build %u.%u",HIWORD(hi), LOWORD(hi), HIWORD(lo),
LOWORD(lo));
```

Example for Visual Basic

```
' Some utility functions
Private Function HIWORD(X As Long) As Integer
   HIWORD = X \ &HFFFF&
End Function

Private Function LOWORD(X As Long) As Integer
   LOWORD = X And &HFFFF&
End Function
```

```
' Return a formatted string with version information
Function GetVersionInfo() As String
  Dim lo As Long
  Dim hi As Long
  Dim result As Long

  result = SSIGetVersion(hi, lo)

  GetVersionInfo = "Version: " & HIWORD(hi) & "." & LOWORD(hi) & ", Build " &
HIWORD(lo)
End Function
```

Using Sort Solution with C/C++

Incorperating Sort Solution into Your Applications Callbacks Structures Error Handling

Files needed

sortsol.dll	The Sort Solution DLL
sortsoli.h	Header file with declarations
sortsol.lib	Import library

Preparing Your Project

1. Add a #include statement for »sortsoli.h« to your source code. If your compiler supports precompiled headers, put it into the header file that creates the precompiled header. For example, if you use Visual C++, put it into »stdafx.h«.
2. Link your project with »sortsoli.lib«

A Complete Example

The following example shows and explains all steps required to execute a Sort Solution profile from within your application.

```
// Include the API declarations. sortsoli.h also includes <windows.h>
#include <sortsoli.h>

// Create a variable of type SSIH which holds the handle to the instance
// The variable "result" is used to save API function results
SSIH handle;
SORTSOL_ERROR result;

// Create a variable of type SORTSOL_CMDFILESTATUS
// zero it out and initialize the "uSize" member

SORTSOL_CMDFILESTATUS cmdstat;
memset(&cmdstat,0x0,sizeof(cmdstat);
cmdstat.uSize = sizeof(cmdstat);

// Create a new sort instance from the profile "mysort.ssp"
result = SSICreateFromFile(&handle,"mysort.ssp",&cmdstat);

// If the instance was created successfully, execute the sort
// by calling SSISort() and free the instance with SSIFree()
if (result == SOSOERR_SUCCESS) {
    result = SSISort(handle);
    SSIFree(handle);
}

// If there was an error creating the instance or during the sort,
// query an error message from the Sort Solution API and print
// it on the screen.
// If everything went OK (usually), display a success message
if (result != SOSOERR_SUCCESS) {
    char msg[255];
    memset(msg,0x0,sizeof(msg));
    SSIGetErrorMessage(result,msg,sizeof(msg));
    printf("%s\n",msg);
}
```

```
else {
    printf("Sort successfully completed!\n");
}

// End of listing
```

You find more examples for the various API functions in the \Samples folder of your Sort Solution installation folder.

Using Sort Solution with Visual Basic

Incorperating Sort Solution into Your Applications Callbacks Structures Error Handling

Files needed

sortsol.dll	The Sort Solution DLL
sortsoli.bas	The declarations for Visual Basic

Preparing Your Project

Add the file sortsoli.bas to your project. This file contains all declarations needed to work with the Sort Solution API in Visual Basic.

A Complete Example

The following example shows and explains all steps required to execute a Sort Solution profile from within your application.

```
' Dim the variables that hold the handle for the sort instance, the
' result codes and the error message (if needed)
Dim handle As Long
Dim result As Long
Dim msg As String

' Create a variable of type SORTSOL_CMDFILESTATUS and initialize
' the member "Size" to the size of the structure
Dim cmdstat As SORTSOL_CMDFILESTATUS
cmdstat.uSize = Len(cmdstat)

' ##IMPORTANT: Set the flag "Override" to false
' if you don't want to override the settings in
' the profile
cmdstate.Override = False

' Create a sort instance from the profile "mysort.ssp"
result = SSICreateFromFile(handle,"mysort.ssp",cmdstat)

' If the instance was created successfully, execute the sort
' by calling SSISort() and free the instance with SSIFree()
if (result = SOSOERR_SUCCESS) {
    result = SSISort(handle)
    SSIFree(handle)
}

' If there was an error creating the instance or during the sort,
' query an error message from the Sort Solution API and print
' it on the screen.
' If everything went OK (usually), display a success message
if (result <> SOSOERR_SUCCESS) {
    msg = String$(255, 0)
    SSIGetErrorMessage(result,msg,255)
    MsgBox msg, , "Error"
}
else {
    MsgBox "Sort successfully completed!"
}

' End of listing
```

You find more examples for the various API functions in the \Samples folder of your Sort Solution installation folder.

KEY DLL's

KEY: User Keys

Sort Solution currently supports over 20 different key types, which is more than enough for to sort nearly any kind of file.

If none of the predefined key types matches your requirements or the records that you have to sort, you can easily add *new key types* to Sort Solution. The key types are named *User keys* because the are user-defined.

To add new key types to Sort Solution, you have to create a DLL containing one or more *Compare functions*. The DLL must follow some simple rules that are required for Sort Solution Key DLL's:

- The DLL must be a 32-Bit DLL for Windows 95 / Windows NT
- The DLL must contain at least one Compare function. Each Key DLL can export any number of Compare functions
- All Compare functions must be exported via the DLL's .DEF file

You can use any number of Key DLL's at the same time and each Key DLL can export any number of Compare functions. This allows you to extend the capabilities of Sort Solution to any level.

The Compare Function

The Compare functions in your Key DLL has to adhere to the following declaration:

```
int WINAPI Compare(void* pKey1, unsigned KeyLen1, void* pKey2, unsigned KeyLen2,
const char* pData);
```

Whenever the Sort Solution engine has to compare two keys, it calls your Compare function with five arguments:

pKey1	Points to the beginning of the first key to be compared. The memory block pointed to by *pKey1* is at least of size *KeyLen1* bytes. The memory area is *not* terminated by a '\0', except the field itself contains a '\0'.
KeyLen1	Size of the memory block pointed to by *pKey1* in byte
pKey2	Points to the beginning of the first key to be compared. The memory block pointed to by *pKey2* is at least of size *KeyLen2* bytes. The memory area is *not* terminated by a '\0', except the field itself contains a '\0'.
KeyLen2	Size of the memory block pointed to by *pKey2* in byte
pData	This argument points to a ASCIIZ string (a string terminated with a '\0'). The content of this string is defined by the <Data> argument of the key type User

The compare function must return on of the following values:

< 0	if Key1 < Key2 (*pKey1 less than *pKey2)
> 0	if Key1 > Key2 (*pKey1 greater than *pKey2)
== 0	if Key1 == Key2 (*pKey1 equal *pKey2)

Example

The following Compare function is extracted from the sample »UserKey« in your Sort Solution sample directory.

The function reassembles the functionality of the key type GENERIC and performs a binary comparison of the both keys.

```
int WINAPI Compare(void* pKey1, unsigned KeyLen1, void* pKey2, unsigned KeyLen2,
const char* pData)
{
```

```
  int res;
  res = memcmp(pKey1, pKey2, KeyLen1 > KeyLen2 ? KeyLen2 : KeyLen1);
  if (res)
    // If the both keys are different, we're finished!
    return res;
  else
    // Else the shorter of the both keys wins
    return KeyLen1 - KeyLen2;
}
```

If you have a key definition like this:

```
KEY(User,ASC,1,2,9,"mykeys.dll", "Compare", "XYZ")
```

and these two records are up to be compared:

```
Ma<b>ximilian</b> Miller;...
Je<b>ff Smart</b>;...
```

Sort Solution will call the Compare function with these arguments:

pKey1	"ximilian"
KeyLen1	9
pKey2	"ff Smart"
KeyLen2	8
pData	"XYZ"

Since the remaining length of the second key is shorter than the definition in the KEY statement (9 byte compared to 8 byte in the key), Sort Solution sets the *KeyLen2* argument to the actual length of the key. The length of the first key is greater than the required key length as defined in the KEY statement, hence Sort Solution uses only 9 bytes of the actual field content for the comparison.

Please study the »UserKeys« example in your Sort Solution \Samples directory for a working example of a KEY DLL.

Exporting the Compare Functions

To use the Compare function with Sort Solution, you must export the name of the function from the DLL's .DEF file. Please note that the name of the Compare functions is **case-sensitive**:

```
LIBRARY      "MYKEYS"
DESCRIPTION  'User-defined keys for Sort Solution'

EXPORTS
    Compare  @1
```

If your DLL contains more than one Compare function, simply add the names of the other functions to the .DEF file.

See also: KEY: User Keys

ASCII Table

This table contains the decimal and hexadecimal codes of the ASCII character set. Lookup the ASCII code for the character you looking for and use the number from the column »Hex« in conjunction with the prefix »0x« in your profiles.

For example, the character code (escape code) for Carriage Return is (10decimal, 0Dhex). To use this code in a profile, you must use a trailing 0x escape sequence: "0x0D".

The combine two or more escape codes in your profile, separate them with a comma:

"0x0D,0x0A" => Carriage Return / Linefeed.

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

Error Messages

The following table contains all error codes that are returned from SORTSOL.EXE and the Sort Solution API.

> The Sort Solution API uses the prefix »SOSOERR_« for all error codes. For example, the error code SUCCESS is named SOSOERR_SUCCESS in the declaration files for C/C++ and Visual Basic.

Code	Code	Description
SUCCESS	0	Successfully completed
UNKNOWN	20001	A unknown error occurred
OPENSOURCE	20002	Error opening the input file Make sure that the file exists and that you have sufficient rights to open it
READSOURCE	20003	Error reading from input file This error indicates a severe physical read error while accessing the input file. When you have a HEADER statement in your profile, make sure that there are at least as many records/bytes in the input file as specified in the HEADER statement
SEEKSOURCE	20004	Error while accessing the input file This indicates a physical read error
CLOSESOURCE	20005	Error while closing the input file This error indicates a severe problem with the file system
PARSESOURCE	20006	Parser error in the input file This error message indicates a wrong file format or a FILETYPE that does not match the format of the input file. Check the input file for inconsistencies and errors and make sure that your FILETYPE specification matches the input file
INVALIDFORMAT	20007	Error in file format Sort Solution has found an incorrect value in the input file. This can happen when the specified file format does not match the file format of the input file or when a key type doesn't match the actual record contents
RUNTIME	20008	Runtime error This is an severe error that cannot be handled by the Sort Solution runtime library. This kind of error should never happen. Please send a message to the program author.
MEMORY	20009	Not enough memory There was an error allocating memory. Reduce the settings for SORTMEM and MERGEMEM in your profile and close some applications
EXPLICITOFFSET	20010	For the file type EXPLICIT. The offset for the length specifier is incorrect or to big.
EXPLICITLEN	20011	For the file type EXPLICIT : The length of the length specifier exceeds the maximum length of five byte
EXPLICITRECLEN	20012	For the file type EXPLICIT : Error while parsing the record length specifier

		This error occurs only when the format of the input file is differs from the FILETYPE specification
ABORTED	20018	Sort aborted This is an informative message
NORECORDS	20019	No records found
		This error occurs when your FILETYPE statement contains an invalid record delimiter. Sort Solution was not able to find a delimiter in the input file and hence did not extract any records
OPENDEST	20020	Error opening/creating the output file
		Make sure that you have enough rights to open/create the output file and that an already existing output file is not *write-protected*
WRITEDEST	20021	Error accessing the output file
SEEKDEST	20022	These errors indicate an physical error while accessing the output file or a disk-full condition
CLOSEDEST	20023	
DESTNOSPACE	20024	Not enough disk space for the temporary files
		Make sure that there is enough disk space on the drives specified with the DRIVES statement or reduce the DRIVESPARE settings
OPENMERGE	20030	Error while creating/opening a merge file
		Make sure that you have write access to all drives specified in the DRIVES statement and that only drives are included that are not read-only. If you don't use a DRIVES statement in your profile, Sort Solution uses all local drives on your computer to distribute temporary files. If you don't have sufficient rights to access all drives, include a DRIVES statement in your profile and exclude drives which are write-protected
READMERGE	20031	Error accessing a temporary file
WRITEMERGE	20032	
SEEKMERGE	20033	This error indicates a physical error on the drive or a disk-full condition
CLOSEMERGE	20034	
GETMERGEFILENAME	20035	Error generating a unique filename This kind of error should never happen. Please send a message to the program author.
NOMERGESPACE	20036	Not enough space for the merge files Delete some files, use the COMPRESS statement to compress the temporary files and reduce the amount of drive space spared with the DRIVESPARE statement
CHECK_SOURCENOTFOUND	20100	Input file not found
CHECK_CREATEDEST	20101	Error creating the output file
CHECK_INVALIDMERGEDRIVE	20102	One or more of the drives specified in the DRIVES statement do not exist or are not accessible
OPENLOG	20120	Error opening/creating the logfile Make sure that you have enough rights to open/create the logfile and that an already existing logfile is not *write-protected*
WRITELOG	20121	Error writing to the logfile
NOKEYSDEFINED	20200	No keys defined
		Sort Solution did not find any KEY statements in your profile or all KEY statements are invalid

ERRORCREATEINSTANCE	20201	Error creating a sort instance This error code is returned by *SSICreate*... when there is not enough memory available to create an additional instance of Sort Solution
ERRORINVALIDERRORCODE	20201	Unknown error This code is returned by SSIGetErrorMessage if the error code passed as the argument is unknown
KEYDLL_NOTFOUND	20300	The key DLL could not be found This code is returned when Sort Solution was unable to find the Key DLL specified in a User key. Verify that the correct name and path of the Key DLL are given
KEYDLL_COMPARE_NOTFOUND	20301	Compare function not found The Compare function specified in a User key could not be found in the specified Key DLL. Make sure that the name is written correctly (**case-sensitive!**) and that the Key DLL exports this function correctly
KEYDLL_NO_NAME	20302	Missing DLL name in a User key
KEYDLL_NO_COMPARENAME	20303	Missing Compare function in a User key
INVALIDMEMORYORPOINTER	20999	Invalid pointer A Sort Solution API function was called with an invalid pointer. Make sure that the pointer is valid and that the memory block the pointer points to is also valid
PARAM_INVALIDHANDLE	21000	Invalid handle An invalid handle was passed to one of the Sort Solution API functions
PARAM_INVALIDPARAM	21001	One of the arguments passed to the API function is invalid
PARAM_INPUTFILENAME	21002	No input file specified
PARAM_OUTPUTFILENAME	21003	No output file specified
PARAM_MERGEDRIVES	21004	No merge drive specified
PARAM_FILETYPE	21005	Unknown type for FILETYPE
PARAM_RECLEN	21006	Invalid or missing record length for file type FIXED.
PARAM_DELIMETERS	21007	The number of delimiters specified is invalid A delimiter must consist of 1 or 2 characters
PARAM_FIELDS	21008	For the file type COUNTED: The number of fields specified is invalid or 0
PARAM_EXPLICITLEN	21009	For the file type EXPLICIT: The specified length is invalid
PARAM_INVALIDKEYTYPE	21010	Unknown key type in KEY statement
PARAM_DATEADJUST	21011	The specified threshold value for the key type Date is invalid. This value must be in a range between 0 and 99
PARAM_KEYPOS	21012	Invalid <Position> for key The <Position> of the key is greater than the number of fields in the record
PARAM_KEYOFFSET	21013	<Offset> greater than record length
PARAM_WRONGFORMAT_ FOR_KEYTYPE	21014	Invalid argument for key type <Position> may only be used for keys when the file type is DELIMITED or COUNTED
PARAM_HEADERSIZE	21015	Size of the header exceeds the input file length
PARAM_FILESIZEMISMATCH	21016	For the file type FIXED : the record length is not a

		divider of the input file size.
PARAM_TOMANYKEYS	21017	To many keys The maximum number of keys allowed in one profile is 64
PARAM_INVALIDMASK	21018	Invalid or missing mask for key type Date or key type Time Both key types require a mask
PARAM_INVALIDSEQUENCE	21019	Invalid sequence or sequence to short in key of type User-defined Sequence
PARAM_INVALIDKEYLEN	21020	Invalid key length The key length for one of the key types Numbers as Strings exceeds the maximum allowed key length
PARAM_DRIVESPACE	21021	Error in CheckDriveSpace statement
VERSIONINFO	21030	Error in *SSIGetVersionInfo*
CMDFILENOTFOUND	22000	Profile not found Check the name and path of the profile
INVALIDCMDFILENAME	22001	Invalid name for profile The filename of the profile is invalid
OPENCMDFILE	22002	Error opening the profile Make sure that the name and path of the profile are correct and that you have sufficient rights to open the file
UNKNOWNKEYWORD	22003	Unknown keyword Check your profile for spelling errors
MISSINGOP	22004	Missing »(«
MISSINGCP	22005	Missing »)«
NODRIVES	22006	Missing drive specifier in DRIVES statement If you want Sort Solution to use all available drives automatically, include no DRIVES statement in your profile
INVALIDDRIVES	22007	At least one invalid drive specifier Check your DRIVES for invalid drive specifiers and make sure that you have sufficient rights to access the specified drives
DRIVESPARE	22008	Invalid or missing argument in DRIVESPARE statement
THREADS	22009	Error in THREADS statement
CACHES	22010	Error in CACHESstatement
SORTMEM	22011	Error in SORTMEM statement
MERGEMEM	22012	Error in MERGEMEM statement
RANGE	22013	Error in RANGE statement
FILTER_ARGS	22014	Invalid or missing arguments in FILTERstatement
FILTER_TYPE	22015	Invalid filter type
FILTER_FTREMOVE_ARGS	22016	Invalid <Remove> argument in FILTER. Only the values TRUE and FALSE are allowed
INPUTFILE	22017	Invalid INPUTFILE statement Check your filename
INPUTFILEDELETE	22018	Invalid <Delete> argument in FILTERFILTER statement. Only the values TRUE and FALSE are allowed
OUTPUTFILE	22019	Invalid INPUTFILEINPUTFILE statement Check your filename
LOGFILE	22020	Invalid INPUTFILEINPUTFILE statement Check your filename

COMPRESS	22021	Invalid or missing argument in COMPRESS statement
HEADER	22022	Missing arguments in HEADER statement
HEADERLEN	22023	Invalid length for HEADER
HEADER_INBYTES	22024	Invalid type for HEADER statement
HEADER_KEEP	22025	Invalid <Keep> argument in HEADER statement Only the values TRUE and FALSE are allowed for this argument
FILETYPE_ARGS	22026	Missing arguments in FILETYPE statement
FILETYPE_UNKNOWN	22027	Invalid type in FILETYPE statement
FILETYPE_INVALIDRECLEN	22028	Record length to big
FILETYPE_SEPARATOR	22029	Invalid or missing separator
FILETYPE_DELIMETER	22030	Invalid or missing delimiter
FILETYPE_TOMANYFIELDS	22031	Number of fields exceeds limit for file type Supported File Types
FILETYPE_EXINVALIDOFS	22032	Invalid offset for file type EXPLICIT
FILETYPE_EXINVALIDLEN	22033	Invalid length for file type EXPLICIT
FILETYPE_EXINVALIDBINSPEC	22034	Invalid type specifier for file type EXPLICIT
KEY_ARGS	22035	Missing arguments in KEY statement
KEY_INVALIDTYPE	22036	Unknown key type
KEY_INVALIDSORTORDER	22037	Missing <Order> in KEY statement
KEY_INVALIDPOS	22038	Invalid <Position> in KEY statement
KEY_INVALIDOFS	22039	Invalid <Offset> in KEY statement
KEY_INVALIDLEN	22040	Key length exceeds limit
KEY_INVALIDDECPL	22041	Missing <Decimal Place>
KEY_MASKTOLONG	22042	Mask to long The mask cannot exceed 100 characters
KEY_MASKTOSHORT	22043	Mask to short The mask must be at least 2 characters long
KEY_SEQUENCETOSHORT	22044	Length of Sequence incorrect A sequence must contain exactly 256 characters
KEY_PRIMLANG	22045	Missing or invalid primary language specifier
KEY_SUBLANG	22046	Missing or invalid secondary language specifier
KEY_DATEADJUST	22047	The specified threshold value for the key type Date is invalid. This value must be in a range between 0 and 99
PRIORITY	22048	Invalid or missing argument in PRIORITY statement
OUTPUTFILEAPPEND	22049	Invalid value for <Append> argument in OUTPUTFILE statement Only the values TRUE and FALSE are allowed
KEYDLL_DATAOUTOFRANGE	22050	The <Data> argument in a KEY statement for the type »User« exceeds the limit of 100 characters
PARAM_TRAILER_WRONGTYPE	22070	Invalid type in TRAILER statement. Allowed are EXPLICIT and PARSED. See TRAILER
PARAM_TRAILER_KEEP	22071	Invalid 'Keep' specifier in TRAILER statement
PARAM_TRAILER_EXPR_TOO_LONG	22072	The tag expression in the TRAILER statement is too long. The expression must not exceed 40 characters
PARAM_TRAILER_LEN	22073	The length specifier for an EXPLICIT trailer is invalid or not numeric
TRAILER_LEN	22080	The trailer is bigger than the input file. Correct the specified length in the explicit TRAILER statement
TRAILER_SAVETEMP	22081	Error while saving the trailer to a temporary file. Sort Solution stores the trailer of your file to a temporary

		file, to be able to handle even large trailers. There was an error while creating the file or during a write operation
TRAILER_READTEMP	22082	There was an error while reading the trailer from the temporary file created during scanning the input file
TRAILER_READINPUT	22083	An error occurred while scanning the input file for th trailer. Check your input file and make sure that you have sufficient rights to access the file

SORTSOL Return Codes

SORTSOL.EXE

SORTSOL.EXE sets the DOS Error Level to one of the values in the following table on return. You can use the DOS ERRORLEVEL statement to analyze the SORTSOL.EXE return code when you execute SORTSOL.EXE from a batch file or a scheduler.

Code	Description
0	Operation successfully completed
1	Missing command line argument. SORTSOL.EXE needs at least the name of the profile to be executed
2	Error while creating a default profile. Possible causes are: • Invalid filename • A profile with the same name already exists and the file is write protected • Not enough disk space for the profile
3	Profile not found
4	Error while reading/interpreting the profile. An diagnostic error message is also printed to the screen
5	Error during sort. a diagnostic message is printed to the screen

Examples

The Sort Solution Script Language , Supported File Types , Keys , Command Overview

This section provides you with a number of comprehensive examples which you can use as a starting point and reference.

All listed profiles contain only the required statements. INPUTFILE, OUTPUTFILE and the like are omitted.

Sorting a Simple Text File

The input file is a simple text file. Each line (aka record) is delimited with a Carriage Return / Linefeed.

```
Smith
McLeod
Miller
McNeally
O'Hara
```

```
FILETYPE(DELIMITED,";","0x0D,0x0A")
KEY(String,ASC,1,0,0)
```

The file format used is DELIMITED. Since in this file each record has only one field, the separator character is not important, so we use a semicolon. The delimiter consists of the hexadecimal equivalents for Carriage Return (13dez = 0x0Dhex() and Linefeed (10dez = 0x0Ahex).

The key definition uses the key type String for field 1. The sort order is set to ASCending. If you want to sort the file in inverse order, simple replace ASC with DESC in the key definition.

Sorting A File Containing International Characters

The following file contains *German Umlauts*.

```
Müller
Schultze
Zubler
Meier
Gröber
Müller
Üssler
Werner
Blöhmer
Groeber
```

To sort this file correct, you need to use Sort Solutions unique NLS support keys:

```
FILETYPE(DELIMITED,";","0x0D,0x0A")
KEY(StringNLS,ASC,1,0,0,LANG_GERMAN,SUBLANG_GERMAN)
```

With this key definition, the strings containing the special German characters are sorted in the correct sequence.

Sorting a File with Multiple Keys

```
3040;München;Meier;06/97;324;10292,43
2045;Frankfurt;Weber;03/87;467;39302,19
1098;Kassel;Kunzert;08/88;1972;126252,72
435;Bonn;Schurz;12/92;2956;27289,99
12;Berlin;Berg;08/89;2923;50560,98
2928;Tokio;Li;09/88;3263;84049,49
1;Brüssel;Zahner;09/88;3293;84049,49
```

To sort this file after the number in the first field, use the following definition:

```
FILETYPE(DELIMITED,";","0x0D,0x0A")
KEY(IntS,ASC,1,0,0)
```

The key type *IntS* interprets the first field as a number and sorts the file after the correct numerical order.

To sort the file after the city name in field 2, use this definition:

```
KEY(String,ASC,2,0,0)
```

Field 4 contains a date in the format MM/YY. To sort the file after this field, use a Date key:

```
KEY(Date,ASC,4,0,0,"MM/YY")
```

If you expect this file to contain dates with a year after 2000, you should use the optional threshold value for the date to make sure that beyond 2000 dates are handled correctly:

```
KEY(Date,ASC,4,0,0,"MM/YY",70)
```

This definition treats all fields with dates earlier than 19**70** as *»2000+«*. 01/07 therefore becomes 01/2007 when sorted.

Field 6 contains a floating point number with the sales of each city/name combination. To sort this field, you must to use a key of type *DoubleS*:

```
KEY(DoubleS,ASC,6,0,0,",")
```

This key type transforms the field content into a floating point number and uses the numerical value to establish the correct sort order. Please note that this file uses a *German floating point format* with *a comma as the decimal place*. The last argument in the key allows you to define the character that should be treated as the decimal place. For American floating point formats you will use a "." instead.

You can combine all or several of these key definitions to create exactly the sort order that you require.

For example, this key sequence sorts the file after the field City, then Dealer and Sales. The resulting file will show the *Top Sales per City per Dealer*.

```
FILETYPE(DELIMITED,";","0x0D,0x0A")
;...City
KEY(StringNLS,ASC,2,0,0,LANG_GERMAN,SUBLANG_GERMAN)
;...Dealer,
KEY(StringNLS,ASC,3,0,0,LANG_GERMAN,SUBLANG_GERMAN)
; Sort: Sales,...
KEY(DoubleS,ASC,6,0,0,",")
```

Sorting a File with a Fixed Record Length

The FIXED format is the easiest format for Sort Solution in respect to sort performance. Since each record has the same length, Sort Solution don't need to parse the file and the records for delimiters. This makes it very fast to read in the file into memory and also to sort the blocks in memory.

```
111122222233334444444445555   // Fields...
191001049601000320.00ABGF
191101049601200430.89BAZF
```

```
1201010596020001200.40GLUG
0756040996140408765.23MMBW
```

Every record of this file is 26 bytes long and contains five fields:

Field1 Integer, 4 bytes

Field 2 Date with the format DDMMYY, 6 bytes

Field 3 Integer, 4 bytes

Field 4 Floating point number with a decimal point and two digits after the decimal place, 8 bytes

Field 5 Generic text field, 4 bytes

This file is sorted with the file type FIXED. Use this statement in your profile:

```
FILETYPE(Fixed,26)
```

Since FIXED format files have no fields in the common sense, all key definitions use offset and length to specify which part of the record builds the sort field. The <Position> parameter of the key definition is always 0.

```
KEY(IntS,ASC,0,0,4)
```

This key sorts the file after the integer number in the first field, starting at offset 0. The length of the field is four byte.

```
KEY(DoubleS,ASC,0,14,8,".")
```

To sort the file after the field 4, use this key definition. The field starts at offset 14 in the record and has a length of eight bytes. The decimal place is indicated with a ».«.

To sort the file in descending order after the text field, use this KEY statement:

```
KEY(String,DESC,0,22,4)
```

Field 2 contains a date with the format "DDMMYY". Use this definition to sort the file after this field in ascending order:

```
KEY(Date,ASC,0,4,"DDMMYY")
```

As always, you can combine as many key definitions to create exact the sequence that you want:

```
KEY(IntS,ASC,0,0,4)
KEY(Date,ASC,0,4,"DDMMYY")
```

These key definitions sort the file after the number in field 1 and within the grouping created by this key, each record is sorted after the date in field 2.

```
KEY(String,DESC,0,22,4)
KEY(Date,ASC,0,4,"DDMMYY")
KEY(DoubleS,ASC,0,14,8,".")
```

Removing Duplicate Records

To remove duplicate records from a file, include a Filter statement in your profile. Depending on your requirements, the filter acts on whole records or on specific fields in each record.

```
3040;München;Meier;06/97;324;10292,43
2045;Frankfurt;Weber;03/87;467;39302,19
1099;Kassel;Kunzert;08/88;1972;126252,72
1098;Kassel;Kunzert;08/88;1972;126252,72
435;Bonn;Schurz;12/92;2956;27289,99
12;Berlin;Berg;08/89;2923;50560,98
1099;Kassel;Kunzert;08/88;1972;126252,72
1;Bussel;Zahner;09/88;3293;84049,49
3041;München;Meier;06/97;324;10292,43
13;Berlin;Berg;08/89;2923;50560,98
```

This file contains three **logically duplicate records**. Logically means that these records refer to the same person although the number in field 1 of the file differs. It also contains one **physically duplicate record** which exactly (byte per byte) matches a second record in the file.

If you want to filter the logically duplicate records, define a key for the *name* and *city* field and a filter with the option KEYS. Again, *StringNLS* is used as the key type to handle the *German Umlauts* in the file:

```
FILETYPE(DELIMITED,";","0x0D,0x0A")
KEY(StringNLS,ASC,2,0,0)
KEY(StringNLS,ASC,3,0,0)

FILTER(DUPLICATES,KEYS)
```

This filter will remove all records from the file that have the same value in the *city* and *name* fields. This will **not** remove the physically duplicate record.

If you want to remove the physically duplicate record, use a filter with the RECORD option:

```
FILTER(DUPLICATES,RECORD)
```

This will only remove the one record in the file that has an exact match with another record in the file.

To remove both kinds of duplicates in the file, use two filters statements:

```
FILTER(DUPLICATES,KEYS)
FILTER(DUPLICATES,RECORD)
```

Logging Duplicate Records

If you want to create a logfile containing all the records removed by your filters, add a LOGFILE statement to your profile:

```
LOGFILE(f:\data\log.dat)

FILETYPE(DELIMITED,";","0x0D,0x0A")
KEY(StringNLS,ASC,2,0,0)
KEY(StringNLS,ASC,3,0,0)
FILTER(DUPLICATES,KEYS)
FILTER(DUPLICATES,RECORD)
```

All records that are removed from your input file due to a filter definition will be written to the logfile. You can view and edit this logfile after the sort.

Sorting Files with Headers

Files that have a special header with additional information at the beginning of the file are quite common. Since it normally makes no sense to sort the contents of the header together with the »normal« records in the file, you need to use a HEADER statement in your profiles for these kind of files.

```
X.DAT Generated 06/23/1997-17.39 Name: MAC
3040;Munich;Miller;06/97;324;10292.43
2045;Frankfurt;Weber;03/87;467;39302.19
1098;New York;O'Hara;08/88;1972;126252.72
435;Tokio;Li;12/92;2956;27289.99
12;Santiago;Santos;08/89;2923;50560.98
```

This file contains a special header record. This record must be ignored when the file is sorted. Since the file is in

delimited format we can use the unit RECORD to define the size of the header:

```
HEADER(1,RECORD,TRUE)
```

This statement skips the first (1) record in the file. The record is written to the output file (TRUE).

If you want to remove the header record, replace the value TRUE with FALSE in the above definition. The record is skipped in the input file and not copied to the output file.

When you have a file with fixed record length, you must define the size of the header in bytes:

```
Y.DAT23069717.39MAC
191001049601000320.00ABGF
191101049601200430.89BAZF
120101059602000120O.40GLUG
0756040996140408765.23MMBW
```

This file has a header with 19 bytes. To handle this header correctly, use this HEADER statement:

```
HEADER(19,BYTE,TRUE)
```

Note: You can also use a header statement to skip the first n records in your input file.

Sorting Files with Trailers

Files which contain an arbitrary number of bytes or records at the end of the file which should be ignored during the sort require a trailer statement in the profile.

To ignore all input file contents that follow the EOF character 1Ahex, include a statement like this in your profile:

TRAILER(PARSED,"0x1A",TRUE)

The TRUE parameter instructs Sort Solution to copy the trailer into the output file after the sort has finished. If you don't want to take over the trailer from the input file to the output file, use FALSE instead.

If your file contains a trailer of a fixed (known) length, you can use an EXPLICIT trailer statement:

TRAILER(EXPLICIT,100,TRUE)

This statement specifies a header of 100 bytes (counting from the end of the file).

If the length of the trailer is unknown, but you know the number of bytes from the input file that should be sorted, use a negative length specifier for the trailer statement in your profile:

TRAILER(EXPLICIT,-100000, TRUE)

This statement tells sort Solution to sort only the first 100.000 bytes from the input file. The rest of the file is treated as a trailer and appended unmodified to the resulting output file.

When you use EXPLICIT trailers, make sure that the specified length does not exceed the size of the input file. Otherwise an error will occur during the sort and Sort Solution will abort.

Using Range Statements

If you want to limit the number of records written to your output file, use a RANGE statement in your profile:

```
RANGE(10)
```

This statement limits the number of records in the output file to 10. Combined with the initial sort this allows you to show only the Top-10 (Flop-10 if you use a descending sort order) records in your output file. Or if you are interested in the first 10.000 records with respect to your key definition, use

```
RANGE(10000)
```

A Complex Profile

The following profile uses all features of Sort Solution to sort a fixed file with a record length of 493 byte. The file has a 36 byte header which is copied to the output file. The file is sorted with three keys, physically duplicate records are removed and written to a logfile.

```
INPUTFILE(a.dat)
OUTPUTFILE(a.out)
LOGFILE(log.dat)

HEADER(36,BYTE,TRUE)

FILETYPE(FIXED,493)

KEY(StringNLS,ASC,0,0,4)
KEY(Date,ASC,0,23,"MM/DD/YYYY")
KEY(DoubleS,ASC,0,89,8,".")

FILTER(DUPLICATES,RECORD)
```

Installation

Sort Solution comes in a self-extracting archive named sosodis.exe. Simply double-click on this file in the Explorer to start the installation.

Enter a directory of your choice in the »Unzip to folder« field. The click on »Unzip« to extract all files from the archive to this directory.

After the files are extracted, you will see a file named »setup.exe« in the directory. Double-click on this file to launch the installation program which will guide you through all required steps to install Sort Solution on your computer.

The install program creates a new entry in your Start menu named Sort Solution.

Files installed

The setup program creates a number of new directories on your computer.

<c:\SortSol>	*This assumes that you have selected c:\SortSol as the install path for Sort Solution*
Bin	Sort Solution command line utilities *SORTSOL.EXE*
Include	Interface and header files for C/C++ and Visual Basic *SORTSOLI.H, SORTSOLI.BAS*
Lib	Library for C/C++ *SORTSOL.LIB*
Licensed	Licensed version of Sort Solution in a password protected archive. Please refer to Shareware and Registration of Sort Solution for further details
Samples	
csmpl	A very simple "C" application to demonstrate how to include Sort Solution into your applications
Tutorial	The tutorial files
MFCSmpl	Full-fledged example on how to use Sort Solution with Visual C++ 5.0 and MFC
Res	Resource files
VCSmpl	Example on how to use Sort Solution with Visual Basic This version uses a callback and therefore runs only with Visual Basic 5.0x
ActiveX	This folder contains examples on how to use the Sort Solution ActiveX control in Visual C++ and Visual Basic
\Windows\System(32)	Binaries, Online-Help and ActiveX control SORTSOL.DLL, SORTSOL.OCX, SORTSOL.HLP, SORTSOL.CNT and SORTSOL.LIC

Uninstalling Sort Solution

To uninstall Sort Solution from you computer, choose the »Uninstall« entry from the Sort Solution menu in your Start menu or open the Control Panel, select the entry »Add/Remove Software«, click on the entry Sort Solution and press the »Uninstall« button.

The Sort Solution ActiveX Control

The Sort Solution ActiveX control (*SortSolX*) is an OLE 2 control that can be embedded in every OLE 2 32-Bit capable application. Is gives you access to the power and flexibility of Sort Solution with an easy to use interface. The Sort Solution ActiveX control is compatible with all of the major development environments (Visual Basic, Visual C++, C++-Builder, Delphi, Power Builder etc.).

The Sort Solution ActiveX has a simple and easy to use interface based on Sort Solution profiles. All information and content of this Online help regarding profiles and their use is also valid for the Sort Solution ActiveX control.

Files needed

sortsol.dll	The Sort Solution DLL
sortsolx.ocx	The Sort Solution ActiveX control
sortsolx.lic	The Sort Solution license file. This file is required only in **development mode**. Don't distribute this file to your customers

Please refer to the documentation of your development environment for more details about ActiveX (OLE) controls and how to use them with your programming environment.

Sort Solution ActiveX topics:

Using the Sort Solution ActiveX control in your projects

Sort Solution ActiveX control property pages

Programming the Sort Solution Control

Please read also the following sections before you start using the Sort Solution ActiveX control:

Sort Solution Technical Backgrounder

Keys

Supported File Types

The Sort Solution Script Language

Using SortSolX In Your Projects

Sort Solution ActiveX Properties Programming the Sort Solution Control

Registering Sort Solution ActiveX

To be able to use the Sort Solution Control in your development environment and your applications, the control must first be registered into you system registry. To do this, you must use the **RegSvr32.exe** utility which is part of Windows 95 or Windows NT.

Open a Command Prompt window and change to the *\Windows\System* (*Windows NT: \Windows\System32*) directory. Then enter and execute the following command line:

```
regsvr32 sortsolx.ocx
```

A message will appear informing you about the successful registration of the Sort Solution OCX.

Now you can use the Sort Solution ActiveX control in every OLE-capable application or development environment.

Note: You can remove the Sort Solution ActiveX control settings from the registry by using the command `regsvr32 /u sortsolx.ocx`

Important:
If you use the Sort Solution ActiveX control in your applications, you have to distribute sortsolx.ocx and sortsol.dll to you customers. You also have to register sortsolx.ocx on your customer's PC's before your application is going to use the Sort Solution control.
Normally this is done automatically by your Install or Setup program (e.g. *InstallShield*).
If you prefer to write you own installation routine, make sure to register Sort Solution correctly by calling *regsvr32.exe sortsolx.ocx*.

Including Sort Solution Into Your Project

Please refer to the documentation of you development environment for more details on how to use ActiveX controls.

Visual Basic

Create a new project or open an existing one. Choose the menu PROJECT and then COMPONENTS. Check the entry *Sort Solution ActiveX control module*.

The Sort Solution icon appears in your toolbox:

Insert an item of this type into your form to use the Sort Solution ActiveX control in your application.

Visual C++ 5.0

Create a new project with OLE ActiveX support (or open an existing one). Choose the menu PROJECT and then ADD TO PROJECT, ... COMPONENTS AND CONTROLS. Choose the entry REGISTERED ACTIVEX CONTROLS from the list and the insert *Sort Solution Control*.

This will create all the required classes and interfaces for Sort Solution in your current project.

See also:

Programming the Sort Solution Control

Sort Solution ActiveX Properties

Sort Solution ActiveX Properties

Programming the Sort Solution Control

This section lists all methods, properties and events of Sort Solution ActiveX.

Properties

Type	Name	Read Write	Description
Files			
BSTR	InputFile	R/W	The name of the input file
BSTR	OutputFile	R/W	The name of the output file
BSTR	Logfile	R/W	The name of the logfile
boolean	DeleteInput	R/W	Set this to *True* if you want to delete the input file after the sort has processed the file See INPUTFILE
boolean	AppendOutput	R/W	Set this to *True* if you want to append the sorted output to an existing output file. If this property is *False*, an existing output file will be overwritten See OUTPUTFILE
boolean	AppendLog	R/W	Set this to *True* if you want to append the output to an existing logfile. If this property is *False*, an existing logfile will be overwritten See LOGFILE
boolean	Override	R/W	If you want to override any of the settings in the profile, set this property to *True* before you call the Sort() method See Overriding Settings from the Profile
Profile			
BSTR	Profile	R/W	The profile to be used by the sort. You can set this property manually in your application or use the method LoadProfile to load an existing profile from disk
Execution			
boolean	Aborted	R/W	Setting this property to *True* causes the Sort Engine to abort the currently running sort
Error Handling (See Error Handling)			
long	Error	R	Returns the last error code
BSTR	ErrorMsg	R	The last error message. This string is empty if no error was reported from Sort Solution (*Error* = 0)
BSTR	ErrorLine	R	If an error was found in the profile, this string contains the full text of that line
long	ErrorLineNumber	R	If an error was found in the profile, this property contains the line number

Statistics (See Analyzing Statistics)

long	StatsSortTime	R	Time for the sort (Phase I) in milliseconds (ms)
long	StatsMergeTime	R	Time for the merge (Phase II) in milliseconds
long	StatsBlockLoadTime	R	Average time in ms to load a block of data (SORTMEM bytes)
long	StatsBlockSortTime	R	Average time in ms to sort a block of data
long	StatsBlockMergeTime	R	Average time to merge CACHES blocks of data (Pre-Merge)
long	StatsNumberOfRuns	R	Number of runs needed for the merge
long	StatsCachePerRun	R	Size of the cache per run in byte (see MERGEMEM)

Methods

`LoadProfile(BSTR FileName)`

This method is used to load an existing profile into the *Profile* property.

FileName contains the name and path of an existing profile.

The method throws an OLE exception if the profile cannot be loaded.

`SaveProfile(BSTR FileName)`

This method stores the profile contained in the *Profile* property into the file described by *FileName*. If a file with this name already exists, it is overwritten.

You can use this method to generate a Sort Solution Profile (.SSP) from your control instance.

The method throws an OLE exception if the file cannot be opened or written.

`Sort()`

This method starts the Sort Engine and executes the sort. If returns after the sort has finished or an error occurred.

This method throws an OLE exception if an error occurs.

`StatsBytesSorted(long* Lo, long* Hi)`

This method returns the number of bytes sorted in form of two 32-Bit values.

Since Sort Solution can handle files bigger than 4 Gigabytes, and OLE lacks the support of 64-Bit data types, the internal 64-Bit variable that is used by Sort Solution must be split into two 32-Bit values. If you are sure that the files you sort are less than 2 GB, it is save to use only the *Lo* value.

`StatsRecordsProcessed(long* Lo, long* Hi)`

This method returns the number of records processed as to 32-Bit values. See above.

`StatsRecordsFiltered(long* Lo, long* Hi)`

This method returns the number of records filtered as to 32-Bit values. See above.

Events

Sort Solution communicates during the sort with OLE events, that are sent to the calling application. The notification method used adheres to the standardized OLE event metaphor and hence all OLE-capable development environments are able handle these events.

Name	Description
SortPercentage	This event is fired during the sort phase (Phase I) to indicate the percentage of completion. The argument of this events contains a value between 0 and 100
MergePercentage	This event is fired during the merge phase to indicate the percentage of completion. The argument of this events contains a value between 0 and 100
BeginSortPhase	Fired at the beginning of the sort phase. The argument of this event specifies the type of sort performed: 0: One-Phase sort 1: Two-Phase sort
FinishSortPhase	Fired at the end of the sort phase
BeginMergePhase	Fired at the beginning of the merge phase
FinishMergePhase	Fired at the end of the merge phase
FinishedSort	Fired after the sort has completed. This is the last event sent

For more information on how to handle the Sort Solution ActiveX events, see Programming the Sort Solution Control.

For general information on how to handle OLE events in your programming language refer to the documentation for your development system.

SortSolX Property Pages

Programming the Sort Solution Control

Sort Solution has as set of built-in property pages which allow you to set and verify most of the properties of the control. A special pane, called the *Executer*, also gives you the opportunity to run a sort directly and test the current profile.

Property Page Profile

This property pages shows you the content of the *Profile* property. It also allows you to create, load, store and edit existing or new profiles.

The ADD button

The ADD button gives you access to the complete keyword hierarchy implemented in Sort Solution. Simply choose one of the keywords to add the corresponding statement to your profile.

Property Page Override

This property pages allows you to access the file-related properties of the Sort Solution ActiveX control. You can use this pane to override settings in your profile or to explicitly define InputFile, OutputFile and Logfile settings that are not included in your profiles.

Important: Please make sure that you check the *Override Profile Settings* box if you want to override settings contained in your profile.

Property Page Executer

The Executer allows you to run the current profile.

Property Page About

This pane displays information about Sort Solution and the Sort Solution ActiveX control.

See also:

Sort Solution ActiveX Properties

Programming the Sort Solution Control

Sort Solution ActiveX Properties

See also: Using the Sort Solution ActiveX control in your projects

Visual Basic

Visual Basic makes it very easy and convenient to use the Sort Solution ActiveX control.

The following assumes that you have added a Sort Solution control to one of the forms in your project and named it *SortSol*.

To load a profile, call the *LoadProfile* method:

```
SortSol.LoadProfile("c:\sortsol\sales.ssp")
```

This call loads the profile "sales.ssp" from disk and initializes the *Profile* property of the embedded control *SortSol*.

You can also set this property manually, by construction a profile in form of a string:

```
Const INPUT_FILE_NAME As String = "sort.dat"
Const OUTPUT_FILE_NAME As String = "sort.out"

SortSol.Profile = "InputFile(" & INPUT_FILE_NAME & ")" & Chr(13) & Chr(10) & _
                  "OutputFile(" & OUTPUT_FILE_NAME & ")" & Chr(13) & Chr(10) & _
                  "Drives(c:)" & Chr(13) & Chr(10) & _
                  "FileType(FIXED,20)" & Chr(13) & Chr(10) & _
                  "Key(Generic,ASC,0,0,20)"

...
```

To execute the sort, simply call the *Sort* method:

```
SortSol.Sort()
```

Error Handling

The Sort Solution ActiveX control reports errors and exceptional conditions through the normal OLE error reporting. You can catch and handle any error that might be thrown be the control with a simple *On Error* statement:

```
On Error GoTo Handler

  ' Load the profile
  SortSol.LoadProfile("c:\sortsol\sales.ssp")

  ' Execute the sort
  SortSol.Sort()

  ' And exit
  Exit Sub

Handler:
  ' Report the error to the user
  MsgBox SortSol.ErrorMsg, , "Error"
```

Handling Events

The Sort Solution ActiveX control uses OLE events to communicate with the calling application. A list of all events fired by the control can be found here.

You can define handlers for the events fired by the Sort Solution control like any other handler for object or control messages. For example, to handle the *BeginSortPhase* event, define a handler like the following in your form:

```
Private Sub SortSol_BeginSortPhase(ByVal SType As Integer)
    MousePointer = vbArrowHourglass
End Sub
```

This handler reacts on the event by changing the mouse cursor to an hour glass. To change the cursor back to the original cursor after the sort has completed, define a handler for the *FinishedSort* event:

```
Private Sub SortSol_FinishedSort()
    MousePointer = vbDefault
End Sub
```

Visual C++

The following assumes that you have a variable named m_SortSol of type *CSortSol* declared as a class member variable.

To load a profile, call the *LoadProfile* method:

```
m_SortSol.LoadProfile("c:\\sortsol\\sales.ssp")
```

This call loads the profile "sales.ssp" from disk and initializes the *Profile* property of the embedded control *SortSol*.

You can also set this property manually from within your application:

```
#define PROFILESTRING "INPUTFILE(input.txt)\r\n" \
                      "OUTPUTFILE(output.txt)\r\n" \
                      "FILETYPE(DELIMITED,\";\",\"0x0D,0x0A\")\r\n"
                      "KEY(String,ASC,1,0,0)\r\n"

m_SortSol.SetProfile(PROFILESTRING);

...
```

To execute the sort, call the *Sort* method:

```
m_SortSol.Sort()
```

Error Handling

The Sort Solution ActiveX control reports errors and exceptional conditions through the normal OLE error reporting. You can catch and handle any error that might be thrown be the control with a *catch*-Handler for *COleDispatchException* or more general with a *CException*:

```
try
{
  // Load the profile
  m_SortSol.LoadProfile("c:\\sortsol\\sales.ssp")

  // Execute the sort
  m_SortSol.Sort()

  ' And exit
  return
}
catch(CException* e)
{
  // Report the error to the user
  e->ReportError(MB_ICONSTOP | MB_OK);
```

```
  e->Delete();
}
```

Handling Events

The Sort Solution ActiveX control uses OLE events to communicate with the calling application. A list of all events fired by the control can be found here.

You can define handlers for all events fired by the Sort Solution control using the ClassWizard. The procedure is the same as for normal messages.

Examples

The Samples\ActiveX directory of your Sort Solution distribution contains full examples for both Visual Basic and Visual C++.