# Using SuperC to Compare or search for data

**SPECIAL: THE COMPLETE STORY ON SUPERC**
**by Donald M. Ludlow**
**IBM Cary Technical Vitality Newsletter**
**2nd Quarter, 1990 – Volume 3, No. 2**

*The developer of SuperC traces the requirements of this powerful compare program and then describes how you can use it – no matter what your job. Whether you use SuperC to increase your productivity or the reliability of your work, SuperC can become a key element in the automation of many manual tasks.*


## Introduction

SuperC is a comprehensive compare and search-for program, not a “C” language compiler. It is both an externally available IBM product and a frequently used internal tool. SuperC versions have been developed for most IBM operating systems. Its objectives are to provide objective use, results, and appearance across all supported environments.

Whether you are a programmer, tester, editor, or a computer specialist (expert or novice), SuperC can increase your productivity in your everyday work. It can improve the reliability pf your work by automating tasks that were previously difficult, or impossible, such as regression testing, change tracking, or searching data. In addition, SuperC can help you compile statistics because it counts changes and lines of source as it compares and searches.

# How SuperC works

**Comparing and file handling**
SuperC provides a power compare capability that inspects data from your input files (data sets in MVS). When you want to know what has been changed between two versions of data, you can use SuperC to detect and display the differences.

Its versatile file-handling capability enables SuperC to process files with large record lengths. It can also process unlimited files sizes with either variable-or fixed-length records.

Providing comparison reports in special highlighted formats is only a small part of SuperC's total capability. SuperC, you can specify options that personalize the comparison. By filtering and changing the input data, you can exclude or change what you do not want (such as header lines, source comments, and changed variable names) and focus on import data differences.

When SuperC completes the comparison, it can produce one report covering the results of the individual file comparison as well as the overall results when you specify a group of files. Minimizing the amount of output as well as generating formats that show you only what you want to see are two of the program's strong points.

**Searching for stings**
Searching files for data strings is similar to filtering the comparison input data. Instead of excluding the data from the search set, SuperC retains the filtered data. It then lists the lines the specified strings or, optionally,provides a count of the number of strings found.

The capability helps you find in which files certain data is used, referenced, or specified (such as names of variables, labels, or a person's address.)

Searching for simple of complex data strings is equally easy. You can extend the search to many files. And, you can also request a number of different reports.

# Before SuperC

**Detecting changes – the manual methods**
If you have used compare programs in the past, you probably found then difficult to use and generally unsatisfying. In fact, you may still prefer to use manual methods; they are often simpler and most often give better results.

Because determining problems, tracking, and keeping records require knowing what has changed, you must still be able to detect differences in changed source, output lists, or any other generated output.

To find differences, you must have two versions of the data, an older one and a newer (changed) one. Then you can use either of the traditional (that is, hard) detection methods; you can work with a split screen within an editor, or you can scan two output listings.

In either case, you use a manual process — gathering the information, marketing changes, and summarizing the changes at appropriate points in the data. the manual process is long, laborious, messy, and error prone. Detecting changes is like proofreading, Sometimes you see them; sometimes you don't/ You can easily overlook even simple changes due to fatigue or inattentive browsing.

If you have ever used the manual methods, you realized that there needed to be a better, simpler, automatic way.

**Detecting Changes – the early automatic methods**
Conceptually, detecting changes between two data files is rather simple. Many early compare programs (now considered rather crude) are still in common use. They can detect whether data files are different, and, if so, they can list the start of the difference. For example, the COMP program is PC/DOS lists up to 10 of the first data mismatches; the infamous IEBCOMPR program on OS/370 provides the same marginal service.

While these all-or-nothing program techniques suffice for detecting basic changes, good change, detection programs should take into account the irregular manner in which most changes occur.

Consider this scenario. Within one program you change two lines, insert an extra routine, detect code at the beginning, and move code from one section to another. You need a compare program than can accommodate all of these conditions.

Finding data matches, skipping over mismatches, and continuing to find data matches has been a failure of most earlier programs. The COMP and IEBCOMPR design, for example, does not include match synchronization after mismatches. Hence, all data after the first uneven number of mismatches between the files get flagged as changed. Both programs fail if the file starts with inserted or deleted code.

## Requirements for change detection programs

Generally, most change detection processes need to start at the top of the both files, can the text data for matching information, and then synchronize after mismatches occur.

This generalization process appears simple, but the process can be more involved. The actual process requires the program to compare a huge amount of data over and over to determine the best compare (or match) set. In addition, the program must allow for duplicate sets of compare sequences, such as PL/I programs that have duplicate DO; or END; lines or sets of comments delimiters. A large number of duplicate data lines and sequences exist in most source programs.

## Design question

A programmed solution must thus answer such design questions as:

- How far must a program search each file before it stops searching from the best compare set?
- What is the minimum size of the match (1,2,3, ….)?
- Do any maximum match set values need to be immediately accepted? (For example, if the program accepts a match set of 10, will it fit overlook a conflicting match set of 20 that it finds later?)
- What happens when a file has large added or deleted sections of data?
- What happens if the first part of either file is missing?
- Can the program give reasonable results when the files have duplicate section?
- If the top of each file is not the best place to start, where should the compare begin?
- What size input records can be accommodated (greater than 256 characters?)?
- Should the compare program handle records for all types of records – variable-length, blocked, with fixed-length attributes?
- Should only complete records be allowed?
- Should sequence numbers be part of the records?
- Should individual sections of the file be handled and how will they be identified?
- Can a big file be compared using the name techniques used in a small file?
- What are the basic units of the compare: lines, words, CSECTs, or files?

The design process must also address a most important objective: who is the intended audience for the compare result output? If the program output is for a person, then the change results should be presented so as to accurately reflect changes. The person should be able to easily interpret the change output listings as indirect indications of the change or editing process. If the changes are to be generated as an intermediate file such as for a library program application, the output could be very cryptic (even encoded) and even less than accurate. Delta differences need only be smaller than the originating source files. The old source plus the delta always accurately yields a copy of the new file.

This last design question, most importantly, dictates the tool process design for comparing and for presenting results.

## Early OS/360 internal compare programs

IBM developed several early internal compare programs that proved to be of some value. These programs could process files that contained minimal changes. And, many times, they produced reasonable results.

Generally, however, users found these programs to be limited and slow. They were limited because the changes had to be simple. The changes also had to fall within the design rules for detecting changes (such as restricting the changes per section to a fixed number with a minimum compare set of some fixed value). They were slow because the programs used a very large portion of time in repetitive string comparing and sorting.

In addition, many of the programs had to write and read auxiliary I/O files to handle storage overflow because of limited size of the processing region.

Most of these compare programs accepted some user input values to achieve better performance. Yet, the generated results often contradicted expected results (at times the programs appeared to be lost!), So, most users did not rely on these programs. Manually counting changes gave more consistent and better results. Automation had not arrived.

## Improved compare programs

**Sequence numbers used for data referencing**
When automation did arrive, the most successful change detection programs were special-purpose programs. They referenced sequence numbers from the sequence columns found on text data lines. By convention, the sequence numbers appeared in a collated order. Thus, the programs counted changes by counting:

- the lines that had different data but the same sequence numbers
- The inserted lines (new sequence numbers not found in the old file).
- The deleted lines (old sequence numbers now found in the new file.).

This process was fast, simple and reliable. Change detection was inherent in the way users stored data. Users needed sequence numbers for references when editing with line editors, such as ISPF, had not been developed yet. This logic for detecting changes worked or many years, and it is still the basis for a major internal distribution library program, CLEAR.

Users knew ti refrain from renumbering program source code and text files. They used only those text editors that did not re-sequence the source file. This convention limited the record types to fixed value (usually 80). thus, users seldom included variable-length records with sequence numbers at the beginning of the data line (such as CLISTSs).

Because program changes were becoming more frequent, users wanted freedom from sequence number restrictions. In addition, they had no convenient way to enlarge the sequence number gaps when inserting large blocks of new source code. Full display editing had arrived.

**Sequence columns unnecessary in new environments**
In many of the newer interactive programs, sequence columns have become relic of the past. Often sequence number columns serve merely as extra columns to hold changing-coding information (such as sequence values with alphabetic product identifies or references).

PC application processes developed that did not allow for sequence numbers columns. Using file saving as the excuse (text lines needed to have blanks in columns near the sequence columns), these processes often omitted the sequence columns and all line-terminating blanks.

These variable records-like formats (used on VM also) became the model for the data files. Little justification existed for using sequence numbers at the end of a line except for comparing and compatibility. Sequence numbers were relegated to the card input era — disaster recovery in some case someone dropped a deck of cards.

**Sequence column dependencies eliminated**
Looking to the future, eliminating sequence column dependencies became a prime design objective for for a site library application program developed in early 1977 at the IBM Lexington site. To meet this objective, the design had to handle successive levels of new source code that had little or no discipline applied to the upkeep of sequence number fields. Users could integrate source from any environment, and the program would detect line differences and track these differences, creating a delta control file in the process. The library program saved only the latest level of the source.

As a substitute for the back-level source code, the program saved a series of compressed deltas. Users needed only to invoke a process that would “reverse apply” the delta to re-create the full source for any back-level code. Considerable space savings resulted from this process because deltas are almost always smaller than the original source. By compressing the delta, the program saved additional space.

Nevertheless, internal IBM library programs, come and go. The Lexington program, although good, was a “home-brew” solution to a problem that needed a more IBM strategic solution.

But, all was not lost from that now-defunct library program. A small integrated component — recognized as having a global value — became a programmer’s utility called SuperC. It has since served many users beyond its original library application.

## Technical basics of SuperC

To recap: SuperC is a compare program that automatically detects differences without using reference number columns. It determines differences based only on the contents of the input data.

Unlike other compare programs that start at the top of two files and find proper pairing of the data, SuperC works differently. It first compresses each input line or word unit into fixed-length (24 bit binary hash total value. Then it uses this smaller compare unit as a substitute for the original variable text compare unit and as the basis for the iterative and heuristic compare process.

Unless both files are an exact match (determined during the initial read pass), SuperC determines matching compare units between each file. It then selects from the candidate list composed of still valid match set to find

the next best sequential matching sets between newly established boundaries. Inserts and deletes are leftover compare units that have no matches between two match sets, such as matches in front of behind.

This process continues iteratively. However, the results must be certified in a later processing pass to confirm that all apparent unit matches agree with the actual record text comparison. An an occasional false match occurrence is corrected in the end results reported to the user.

This process works relatively fast because SuperC makes all intermediate comparisons by using the short, 14-bit hash sum value. This hash sum represents every line (line compare), word (word compare), or byte (byte compare), which stretches each 8-bit byte to 24 bits) in the files. SuperC can also use the 24-bit, fixed=length hash sum values as indexes into arrays. This simplifies the process for identifying which compare units from each file match.

## SuperC environments

SuperC operates in MVT, MVS, VM, PC/DOS. OS/2, OS/400, and their OS38 environments. Available in shipped IBM products, extended versions of SuperC are also available from the IBM internal tools disks. v is currently a series of IBM programs maintained and owned by the ISPF/PDF area within the Cary laboratory.

You might have seen SuperC implemented a series of panels that allow you to enter various information and options. Or you might have invoked it directly from a command line, such as FULIST in VM or FileCommand in PC/DOS. Other programs can invoke it as an embedded service function to generate database differences for later transmissions to master nodes. CMPPART in IDSS and a service routine within CALLUP use SuperC (or SuperC — a specialized subset of SuperC) in this way.

## Functions

SuperC provides six important functions that allow you to control the compare process.

**Different levels for comparing data:** SuperC compares data at the following levels:

- File level (comparing entire files to other files)
- Line level (comparing individual lines between files)
- Word level (comparing an individual sequence of words appearing on lines)
- Byte level (comparing sequences of individual bytes within a line).

If you are like many users, you might compare only lines (the program default). However, you can use file compare as a fast method for checking which files are the same or different. Comparing bytes works best for binary data files.

**Different levels of listing reports:** You can request the following reports:

- Long, or complete, report (with changes)
- Delta report (changes only)
- Change report (matches before and after the changes)
- Overall summary report (statistics only)
- Side-by-side resports (both files displayed in adjacent columns)
- Change-bar reports (flagging only changed lines)
- No listing (compare results indicated by return codes)

All reports have a summary section listing the statistics for change activity, file size, and options used. Some users run SuperC only to obtain this summary information.

**Different output formats – listing or control file:** You can request SuperC to return the results in two different formats.

- Listing output (meant to be printed and or browsed online in a report format)
- Control file (up to eight different control file formats, some with source data that is easier to be interrupted and processed by a user-written postprocessing program)

**Input filtering:** You may prefer preprocessing the input files (separately or by using SuperC operations) before comparing them. With SuperC preprocessing options you can exclude unwanted lines, such as headers or comments that you changed during input. By filtering, you can focus better on the important changes.

**Output filtering:** The following process options minimize the output volume of the generated reports, or they personalize the output information:

- REMOVR option — Overrides the reformatted line status in the change listing so reformatted lines appear as matched lines.
- LOCS option — Lists only changed members in the summary section. This can reduce the number of isted members in the oarticitioned data set (PDS) or MACLIB summary for compares that have few changes.
- NOPRTCC option — Suppresses output page separators to make the appearances olf the report more readable.

**Soncolidation of results into a combined report:** You can specificy certain options that generate one report with the results from many compares, such as:

- Multiple members from one file:
  VM–MACCLIBs (VM)
  MVS–PDSs (MVS)
- Multiple files per execution:
  MVS–Concatenation of files
  VM–SELECTF (select files statements)
  VM/DOS/OS2 — Wild charatcers (*) files names
- Appending a listing to the end of previous output results.

# Use within IBM

Some of the ways you can use SuperC follow:

**Maintaining libraries:** You can use SuperC (or SuperCU) to generate delta listing and provide specialized update files for maintaining library control.

**Participating in programming inspection:** SuperC's side-by-side listings highlight changes; its long or change listings detail complex module changes. Thus, you can yse these highly readable listing to see changes between the current source code and the previous level.

**Interacting with other programmers (using separate libraries):** When two groups work on a common project, code can cbe shared or separate. In wither case, each group must be aware of changes that affect the other.

Programmers can track both common code and separate auditable code by using a data link to communicate updates. They can exchange deltas, updated source code, or SuperC listings with annotated changes. Both groups can then detect any changes that affect them and update theor local library database.

**Merging two independent changes:** When two groups work on a common source module, they can directly edit the control file generated from the two versions of the common code (using UPDLDEL option). They can accept changes as a whole (a block of inserts), a combination of changes (a portion of the inserts with some deletes), or a revision of the changed source (the intersecting code changes).

**Leeping vital records of changes:** Some of the data processing centers require hard-copy verification before accepting changes to or promoting source code for vital programs. SuperC listings provide an accepted accurate record of changes (whether those changes are intentional or unintentional).

**Facilitating regression testing:** Because SuperC can compare most program output, you can test successive levels of a program by monitoring the output. You can trace differences in the output to changes areas in the program and then check them for the expected results.

During regression testing you usually process large numbers of test cases, producing voluminous output (many times with only small differences). SuperC easily handles large numbers of records. The combination of a generalized record format, variable data content, resynchronization after mismatching section, and quick processing make SuperC superior for this task.

SuperC is an alternative to more specialized test-checking programs that are often faster but are limited to whatever assumptions the designers made about how data is arranged. SuperC might be better for preliminary tests before vesting sources to design the specialized programs.

**Generating management reports and program statistics:** UYou can use SuperC to:

- Count lines of code.
  Use this count in making productivity projections. (Productivity projections are usually based on the number of lines of code, program complexity, and the programming language.)
- List number of changes to establish programs and modules directly affects maintenance costs.
- Count comments.
  Because SuperC can recognize certain programming language comments, you can subtract comments from lines of code. You can filter them from any compare set to determine changes, or you can count them independently.

**Ensuring security with deltas:** By saving changes files in a delta format, you can apply the deltas to upgrade the original database to new levels of function. Hence, communicating deltas requires fewer security precautions, as long as the original database is secure (such as, purchased). Contrast that with the problem of updating a function by communication the entire, newly updated database and with the problem of checking authorization.

**Assuring quality:** You can use SuperC to quantify the change activity of any new software product release. Use the number of programs changes as one criterion to predict the future error rate.

# User application

Whatever you job, you can use SuperC in the following specific ways.

**Writers and editors** can:

- Detect word changes within documents.
  – SuperC finds word differences even if the words have been moved to adjacent lines
  – One option (UPDREV) allows you to have SuperC generate SCRIPT header delimiters around changed SCRIPT text files lines. Subsequently, SCRIPT processes this modified copy of the user source file causing all changed lines to be flagged in the left column of the script output.
- Verify that only designated areas are changed.
  – Because the comparison results show all areas affected, changes made to restricted areas might be invalid. Therefore, SuperC can detect unintended changes so that you do not need to check the completed document for errors again.

**Programmers and system administrators** can:

- Compare two files that have been reformatted.
  – Reformatted files contain such differences as changes to indentation levels and spaces inserted or deleted.
  – SuperC detects and classifies reformatted lines as special changes. You can request that these lines be listed in the output along with the normal insert and delete changes, or you can request that they be eliminated from the listing. Reducing the number of flagged lines help you focus on important, rather than cosmetic changes.
- Determine whether two files have corresponding, like-named members.
  – SuperC lists any members absent from one file but present in the other in addition to all change activity between like-named members.Thus, you can see changes caused by creating or deleting members within the file.
- Generate management reports that show the number and type of changes in source code.
- Use comparison results to summarize changes.
  – Because SuperC can count the changes and unchanged lines of code in program, you can use the comparison results to summarize the changes between different version of the program.
- Retain a record of change activity.
  You can collect SuperC's listing files and retain them as a permanent record of the changes made to a new program before it is released. Source code differences can help you detect regressions or validate the appropriateness of any modifications.
- Compare files developed in unconnected systems.
  – Because file compares generate 32-bit hash sum value and count the number of bytes and lines for each file or member compared, you can use the hash sum values developed on different systems. Files compared on two different systems having different hash sum values are never identical; files that have identical hash sum values have a high probability of being identical. You can also check the number of lines and number of bytes as secondary indicators.
- Develop additional uses for the SuperC process files.
  – SuperC produces general results with generalized output reports. However, you may have unique requirements. If so, use the process options UPDCNTL, UPDSEQ0, and UPDLDEL to provide comparison data that can be processed by other programs. These files follow a fixed output format, and they are relatively easy to use for postprocessing.

## IBM application that use SuperC

**Telephone directory program (CALLUP)**: This program used a scaled-down version of SuperC (referred to as SuperCU) to assist in updating the internal telephone directories. This version could merge the directory changes into each previous directory's database.

The programmers invoke SuperCU to generate a special update file consisting of directory changes at remote sites. The automatic process sends the delta across the internal network to a central repository.

The central site receives the files, invokes another option within SuperCU that checks the hash sum values on both the original and updated directories (carried in the delta file), and reverse applies the deltas. The hash sum values ensured that the directory database is consistent with the originating directory both before and after the update.

Using SuperCU to detect the directory changes and to send the updated file also results in less data being transmitted over the network. The alternative would be to send the entire updated directory database.

With SuperCU's interchangeable format for change tracking, the application uses the updated file as a compression technique yet this ensures secure regeneration. (However, this only works if the receiving location has an established database.)

**Generalized simulator application:** This program, an IBM internal test tool, can debug certain programs that contain computer instruction statements.

Normally, each time a tester makes a change to update the simulator logic, regression tests are run to check the results of the standard generated test output. The tester then visually inspects each detected change to ensure that only the intended results were generated.

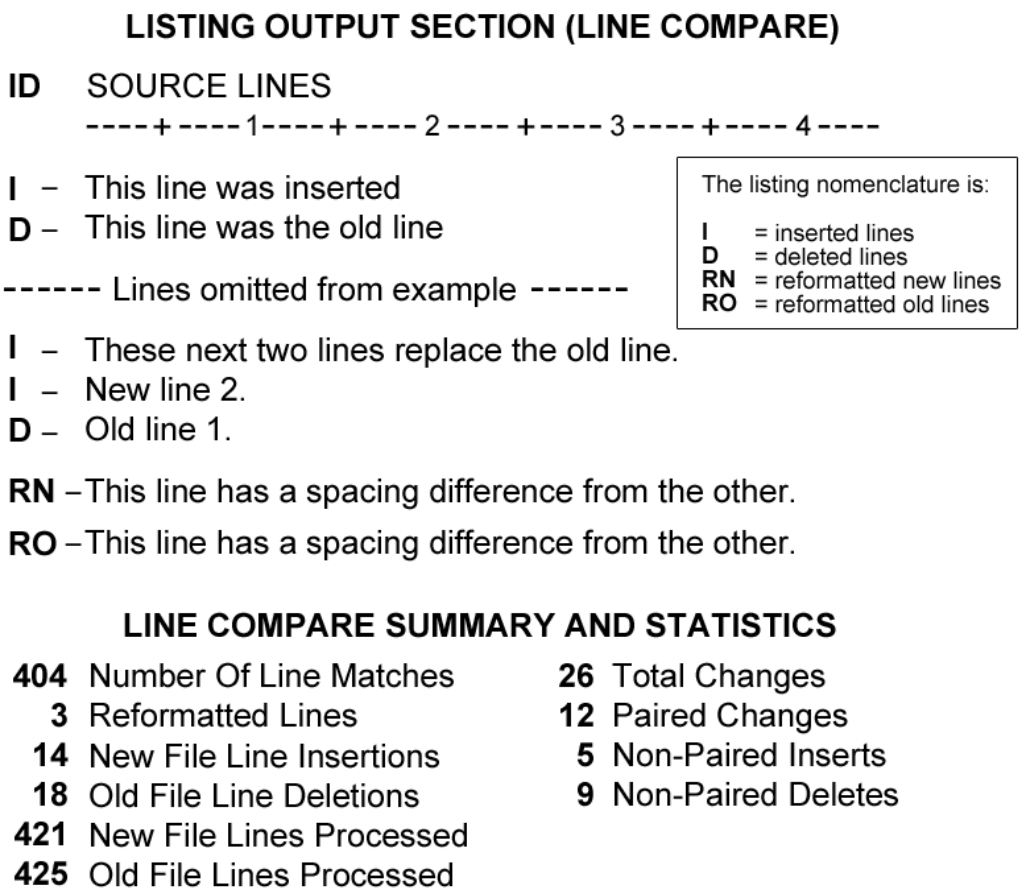
The simulator test bucket for this test tool generated about 25,000 lines register and storage data. to visually inspect the entire output file to verify where changes occurred and whether they were correct would be an enormous task. Instead, the testers used SuperC to compare the test case results from the previous level of the simulator. SuperC produced a delta output file results were often reduced by a factor of 100-150. With only 200-300 lines needing to be checked, the testers could visually inspect the changes much easier.

The testers often staged changes to the simulator so that the output deltas could be reduced to an incrementally smaller number for each changed increment. This made the regression review process easier, also.

**Configurator:** A product group built a system configurator so that customers could cutom-order its product. The developers used SuperC to compare the customers' configurations against the valid configurations. The masters output file, a consolidated configurator file, contained the valid configuration. They used the CMPLINE statement to orient SuperC to the proper section of the configurator file.

The configurator test plan called for approximately 800,000 SuperC comparison runs. Estimates for the number of SuperC compares for a subsequent release were projected at 1.5 million.

**Figure 1.** Example of a SuperC delta listing



LISTING OUTPUT SECTION (LINE COMPARE)

ID SOURCE LINES
----+----1----+----2----+----3----+----4----

I – This line was inserted
D – This line was the old line

------ Lines omitted from example ------

I – These next two lines replace the old line.
I – New line 2.
D – Old line 1.

RN – This line has a spacing difference from the other.

RO – This line has a spacing difference from the other.

The listing nomenclature is:

I = inserted lines
D = deleted lines
RN = reformatted new lines
RO = reformatted old lines

LINE COMPARE SUMMARY AND STATISTICS

| | | | |
|---|---|---|---|
| 404 | Number Of Line Matches | 26 | Total Changes |
| 3 | Reformatted Lines | 12 | Paired Changes |
| 14 | New File Line Insertions | 5 | Non-Paired Inserts |
| 18 | Old File Line Deletions | 9 | Non-Paired Deletes |
| 421 | New File Lines Processed | | |
| 425 | Old File Lines Processed | | |

**Example of compare output**
Figure 1 shows an example of a simplified delta output listing based on invoking SuperC, requesting a line compare, and producing a delta output report. The end of the listing shows the compare summary statistics.

For a line compare, SuperC reports the differences as possible modification to the old file to create the new file. It reports differences as actions, lines are deleted from the old file, and lines are inserted in the new file. It also detects other differences, such as a line in which the text on the line is reformatted (for example, a special insert and delete pair with spacing differences).

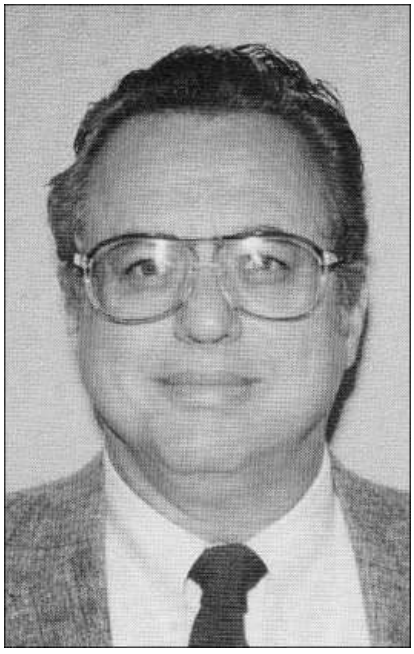
**In conclusion**
You now have many suggestion for ways to use SuperC. However, do not be limited by these suggestions. You can apply its many capabilities in an endless variety of application.

Try it!

---

***Don Ludlow**, Senior Technical Staff Member, currently focuses on planning, designing, and implementing new features in SuperC for ISPF/PDF and other IBM product areas.*

---



Don Ludlow currently works for R.C. Miller in AD Product Development, but his main focus is the development of SuperC, a software compare program. SuperC was first released as an IBM System/370 MVS/VM and IBM PC product and later integrated as part of ISPF/PDF V2.3 utilities.

Don Ludlow joined Poughkeepsie in 1962 as an associate programmer. During his career he has held various technical and managerial positions in programming, computer design, and systems architecture. He transferred to RTP in 1984 from Lexington, KY and to Cary last October.

He has received three IBM Outstanding Achievement Awards, an Outstanding Innovation Award, and an IBM Excellence Award. He received an Outstanding Contribution Award in 1967 for early OS/360 system design efforts. Son made numerous disclosure submissions with several being published. He holds two U.S. patents.

Don was the principle designer and developer and later manager of the Input-Output Supervisor (IOS) component of OS/360. Later he headed PLS/85 and PLS/86 compiler development. He also developed SUPERZAP (an FE data change support tool).

How did he get to where he is today? Don's answer:

> *"With a positive attitude, you can accomplish anything. It's all in the "mindset." Identify problem areas and find good sound solutions. Be willing to take a chance, put yourself on the line, and get management support; that's a good start.*
>
> *I preach the "put your arms around it and love it" principle. There's nothing wrong with admitting you love what you are doing and working extra hard at it. You own your assignment until it's time to move on. It's too easy to approach tour work as a 9-to-5 chore. Be different. Get excited. Quality and innovation follow as a result of hard work, dedication, and perseverance. Remember all hard work doesn't get recognized. You also have to be a team player."*

* * * * * * *

**https://donludlow.com/using-superc-to-compare-or-search-for-data/**